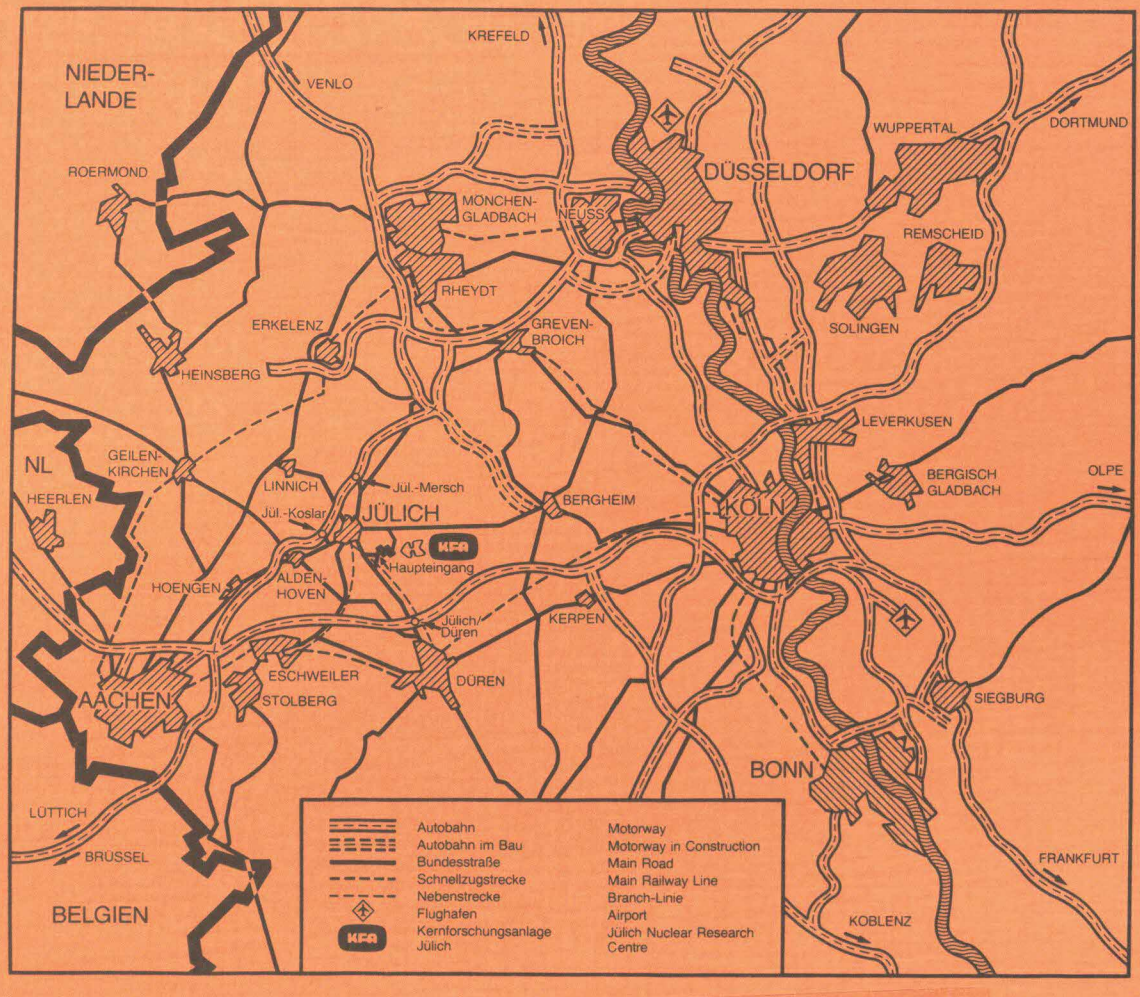
KERNFORSCHUNGSANLAGE JÜLICH GmbH
Zentralinstitut für Angewandte Mathematik

Ein Software-Paging-System für die Bearbeitung sehr großer Matrizen in FORTRAN

von

Ulrich Detert

Jül - Spez - 372
September 1986
ISSN 0343-7639



Als Manuskript gedruckt

Spezielle Berichte der Kernforschungsanlage Jülich – Nr. 372

Zentralinstitut für Angewandte Mathematik Jül - Spez - 372

Zu beziehen durch: ZENTRALBIBLIOTHEK der Kernforschungsanlage Jülich GmbH

Postfach 1913 · D-5170 Jülich (Bundesrepublik Deutschland)

Telefon: 02461/610 · Telex: 833556-0 kf d

Ein Software-Paging-System für die Bearbeitung sehr großer Matrizen in FORTRAN

von

Ulrich Detert

Abstract

Many today's supercomputers exhibit a certain mismatch between computational power on the one hand and memory capacity on the other hand.

This paper presents a fast and flexible method for handling large FORTRAN data arrays - especially 2-dimensional matrices - by a software paging mechanism originally designed for use with CRAY mainframes. The paging mechanism uses matrix segmentations with user-defined shape and size of blocks. The user interface is realized by a set of FORTRAN callable subroutines providing access to all matrix substructures.

In this paper the principle of operation of this method is presented. Performance data for various types of matrix segmentations are given and two page replacement strategies are compared. Furthermore a comparison of this method with a similar method found in literature is given.

Together with the CRAY version a portable version written in standard FORTRAN 77 is provided. This version is mainly intended to be used on IBM mainframes.

In the appendix a short user's guide gives a summary of all routines contained in the software paging method. Some sample program listings of typical routines are also provided.

Vorwort

Diese Arbeit ist aus dem Bedürfnis heraus entstanden, für große Anwenderprogramme auf der CRAY X-MP der KFA eine flexible Methode zur Nutzung großer externer Datenspeicher zu haben.

Allen, die durch Ideen, Diskussionsbeiträge und die Bereitschaft zum Testen des erstellten Systems zu dieser Arbeit beigetragen haben, gilt mein Dank für ihre Hilfe.

Insbesondere danke ich Herrn H. W. Homrighausen für die Anregung zu dieser Arbeit, Herrn Dr. B. Steffen für wertvolle Ideen zum Konzept der Blockzerlegung, Frau B. Lange für die Durchführung von Messungen an FORTRAN-Programmen und Herrn W. Nagel für intensives Korrekturlesen. Desweiteren danke ich der Firma CRAY Research GmbH für die freundliche Bereitstellung von Informationsmaterial über die CRAY I/O-Prozeduren.

Mein besonderer Dank gilt dem Institutsleiter des Zentralinstituts für Angewandte Mathematik, Herrn Prof. Dr. F. Hoßfeld.

Inhaltsverzeichnis

Beschreibung des Software-Paging-Systems

Einleitung

Für viele Aufgabenstellungen aus dem Bereich der rechnerunterstützten Forschung (Computational Science) sind zwei Hauptanforderungen an die verwendeten Rechner typisch:

1. hohe Verarbeitunggeschwindigkeit bei arithmetischen Operationen,

2. große Mengen an wahlfrei adressierbarem Hauptspeicher.

Viele der am Markt erhältlichen Höchstleistungsrechner weisen jedoch ein gewisses Mißverhältnis zwischen Rechengeschwindigkeit einerseits und Größe des verfügbaren Hauptspeichers andererseits auf. Für viele Anwendungen bedeutet dies, daß Problemdimensionen, die bezüglich der Rechenzeit in vertretbaren Zeiträumen bearbeitet werden könnten, oft nur mit erheblichem Aufwand realisiert werden können, da die auftretenden Datenmengen nicht vollständig im Hauptspeicher gehalten werden können.

Typische Beispiele für solche Anwendungen sind Programme aus dem Bereich der theoretischen Physik - insbesondere der Vielteilchenphysik und der Elementarteilchenphysik - der Seismik und der Meteorologie. Typische Lösungsmethoden sind Simulationsrechnungen auf diskreten Gittern, Lösungsverfahren für lineare Gleichungssysteme und Multi-Grid-Verfahren zur Lösung partieller Differentialgleichungen.

Viele dieser Programme weisen im Kern als Datenstrukturen zwei- oder mehrdimensionale Felder auf, die vor allem im Falle vektorisierender Algorithmen verhältnismäßig regelmäßig durchlaufen werden. Daher liegt es nahe, bei der Implementierung solcher Anwendungen für die Datenspeicherung virtuelle Speicherkonzepte zu verwenden. Die Programme arbeiten dabei logisch gesehen auf einem "im Prinzip unendlichen Speicher", der technisch gesehen durch schnelle Sekundärspeicher (z.B. externe Halbleiterspeicher oder Magnetplatten) realisiert wird.

Das im folgenden beschriebene Software-Paging-System wurde originär für die CRAY X-MP entwickelt und hinsichtlich der Leistungsdaten für diesen Rechner optimiert. Daneben steht jedoch auch eine portable Version zur Verfügung, die insbesondere auf den IBM-Rechenanlagen der KFA lauffähig ist. Beide Varianten sind in FORTRAN geschrieben; die CRAY-Version enthält jedoch nicht-standardisierte Spracherweiterungen, während die portable Version vollständig in FORTRAN 77 geschrieben ist.

Im folgenden soll zunächst das Konzept des Software-Paging-Systems näher erläutert werden. Hieran schließt sich eine Beschreibung der daraus resultierenden Adressierungsmethode an. Ein wesentliches Augenmerk wird schließlich auf die Effizienz des Verfahrens gerichtet. Hierbei wird zunächst der Einfluß der Matrixzerlegung auf die Effizienz untersucht; weiterhin werden zwei Pufferverwaltungsstrategien für das Paging-Verfahren miteinander verglichen. Abschließend werden dann die CRAY- und die IBM-Implementierung mit einem funktional ähnlichen Verfahren aus der Literatur verglichen.

Soweit im folgenden Meßwerte von Programmläufen wiedergegeben werden, entstammen sie Rechnungen, die in der normalen Produktionsumgebung des ZAM durchgeführt wurden. D.h. die Meßwerte sind nicht frei von Störungen, die durch andere Benutzerprogramme hervorgerufen werden können. Dieser Tatbestand ist besonders bei den für manche Messungen zum Ein-/Ausgabeverhalten angegebenen I-/O-Wait-Zeiten zu berücksichtigen, die extrem umgebungsabhängig sind. Die Messungen auf der CRAY X-MP des ZAM wurden unter COS 1.13 mit der Compiler-Version CFT 1.13, die Messungen auf IBM auf einer IBM 3081 unter VM/CMS, HPO 3.4 mit VS FORTRAN 1.4.1 durchgeführt.

Konzept

Bei herkömmlichen virtuellen Speichersystemen mit Seitenaustausch-Verfahren (Paging) werden die Datenbereiche eines Anwenderprogramms in der durch die Programmiersprache vorgegebenen Weise gespeichert, die Speicherbereiche in Seiten fester Länge aufgeteilt und im virtuellen Speicher (externes Speichermedium) abgelegt. Für in FORTRAN gespeicherte Matrizen bedeutet dies wegen der spaltenweisen Abspeicherung, daß z.B. Matrixspalten auf wenigen zusammenhängenden Pages abgelegt werden, während Matrixzeilen auf viele Pages dünn verteilt gespeichert sind. Im Extremfall hat dies zur Folge, daß beim Zugriff auf eine Matrixzeile sämtliche Pages einer Matrix referiert werden. Da bei der Software-Realisierung eines Paging-Systems der Datenzugriff nur über wohldefinierte Unterprogrammaufrufe erfolgt, kann man sich von der Notwendigkeit der spaltenweisen Speicherung freimachen.

Bei dem im folgenden beschriebenen System wurde eine Blockzerlegung der zu speichernden Matrizen gewählt. Größe und Form der Blöcke können vom Anwender frei definiert werden, mit der einzigen Einschränkung, daß eine regelmäßige Zerlegung der Matrix in Blöcke gleicher Größe gegeben sein muß. Jeder Block der Matrix wird als ein Datensatz auf dem externen Speichermedium abgelegt. Auf diese Weise ist hohe Flexibilität und Effizienz beim Datenzugriff erreichbar. Blöcke können z.B. aus einer oder mehreren Matrixzeilen oder -spalten bestehen, um den Zugriff auf Matrixzeilen bzw. -spalten zu begünstigen; Blöcke können aber auch quadratische Teilmatrizen der Matrix sein, wodurch gleiche Effizienz für den Zugriff auf Zeilen und Spalten erreicht wird (s. Abbildung 1 auf Seite 3).

Die grundsätzliche Organisation des Software-Paging-Systems ist damit die gleiche wie von herkömmlichen Paging-Verfahren bekannt. Die zu verarbeitende Matrix liegt, in Blöcke aufgeteilt, in ihrer Gesamtheit auf dem externen Speichermedium (Virtueller Speicher); Teile der Matrix, die gerade bearbeitet werden, befinden sich in einem Datenpuffer im Hauptspeicher ("Working Set"). Die Verwaltung des Puffers (Nachladen neuer Blöcke, Verdrängen nicht mehr benötigter Blöcke) übernimmt das Unterprogrammsystem je nach den Erfordernissen (s. Abbildung 2 auf Seite 4).

Im vorliegenden Fall wurde ein solches virtuelles Speicherkonzept speziell für zweidimensionale Matrizen realisiert. Der Datenzugriff erfolgt durch Unterprogrammaufrufe, die jeweils Teilstrukturen der gespeicherten Daten verfügbar machen. Bei der Auswahl der ansprechbaren Teilstrukturen wurde besonderes Gewicht auf die Berücksichtigung vektorisierender Sprachkonstruktionen gelegt. Die Vektorisierung bleibt auch bei Benutzung des Unterprogrammsystems erhalten.

Im einzelnen kann auf folgende Matrixteilstrukturen lesend und schreibend zugegriffen werden:

- Matrixzeilen (oder Teile davon),
- Matrixspalten (oder Teile davon),
- Diagonalen bzw. Nebendiagonalen (oder Teile davon),
- Blöcke (rechteckige Teilmatrizen),
- Matrixelemente.

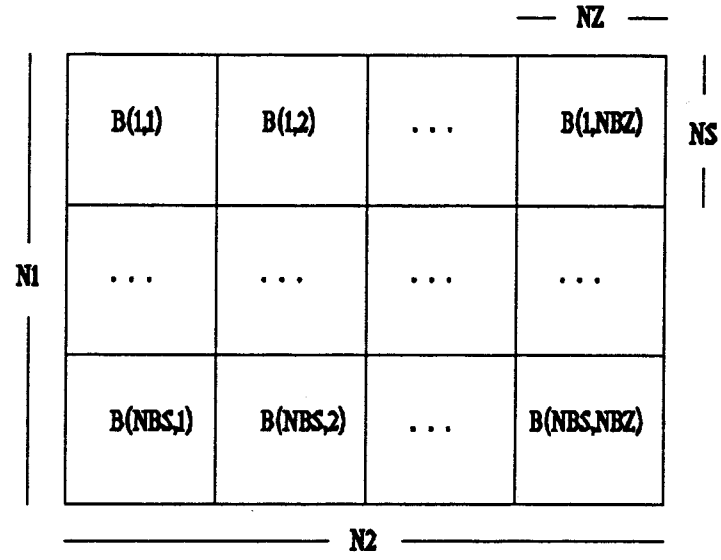


Abbildung 1. Blockzerlegung

Adressierung der Blöcke

Sei eine gegebene $N_1 \times N_2$ -Matrix allgemein in Blöcke der Größe $N_S \times N_Z$ aufgeteilt. Die Matrix enthält dann

$$\frac{N_1 N_2}{N_S N_Z}$$

Blöcke. (Ist N_1 nicht durch N_S teilbar bzw. N_2 nicht durch N_Z, so kann die Matrix ohne große Effizienzeinbußen durch Hinzunahme von maximal $N_S - 1$ Dummy-Zeilen bzw. $N_Z - 1$ Dummy-Spalten so erweitert werden, daß eine Blockzerlegung möglich ist).
Es müssen

$$M_S := \frac{N_1}{N_S}$$

Blöcke angesprochen werden, um auf eine Spalte der Matrix zuzugreifen bzw.

$$M_Z := \frac{N_2}{N_Z}$$

Blöcke, um auf eine Matrixzeile zuzugreifen.

Für den Zugriff auf Diagonalen werden maximal

$$M_S + M_Z - 1$$

Blöcke benötigt.

Die durch die Blockzerlegung gegebene Blockmatrix sei definiert als

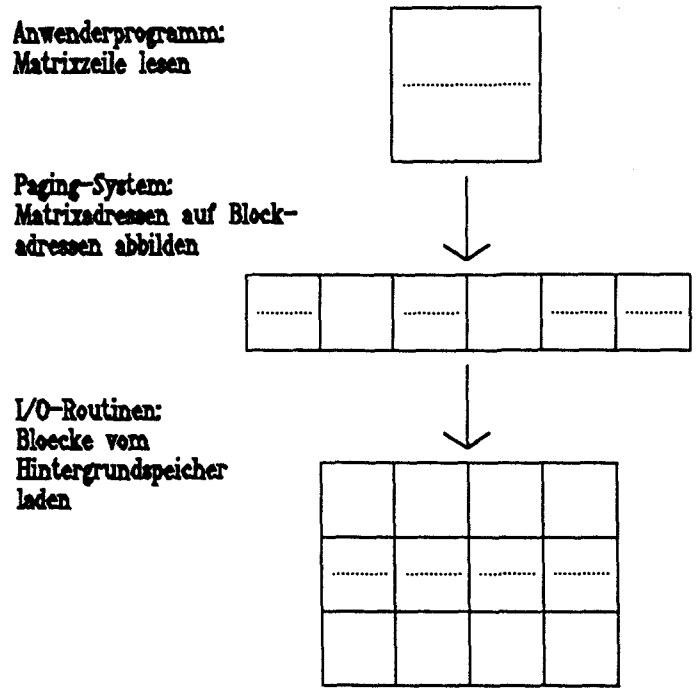


Abbildung 2. Virtuelles Speicherkonzept

$$\mathbf{B} := (\mathrm{B}_{i,k}),\ 1 \le i \le M_S,\ 1 \le k \le M_Z, \tag{1}$$

mit

$$\mathrm{B}_{i,k} := (b_{j,l}),\ 1 \le j \le N_S,\ 1 \le l \le N_Z. \tag{2}$$

Damit liegt das Matrixelement $a_{i,k}$, $1 \le i \le N_1$, $1 \le k \le N_2$ im Block

$$\mathrm{B}_{\lfloor (i-1)/N_S \rfloor + 1,\ \lfloor (k-1)/N_Z \rfloor + 1} \tag{3}$$

und hat in diesem Block die relative Adresse

$$b_{(i-1 \bmod N_S)+1,(k-1 \bmod N_Z)+1}. \tag{4}$$

Für den Zugriff auf die i-te Zeile der Matrix werden die Blöcke

$$\mathrm{B}_{\lfloor (i-1)/N_S \rfloor + 1,\, n},\ 1 \le n \le M_Z \tag{5}$$

benötigt. In jedem Block ist die Zeile

$$b_{(i-1 \bmod N_S)+1,m},\ 1 \le m \le N_Z \tag{6}$$

anzusprechen. Die Adressierung von Spalten der Matrix erfolgt analog.

Für die Adressierung von Diagonalen und Nebendiagonalen der Matrix gilt folgendes:

Sei

$$\mathrm{D}^{r,t} := (d_{r+k,t+k})_{0 \le k \le \min(N_1 - r, N_2 - t)}$$

eine Diagonale oder Nebendiagonale der Matrix.

Die Adresse des Blocks, der das erste Diagonalelement $d_{r,t}$ enthält, berechnet sich aus (3); entsprechend ergibt sich die relative Adresse dieses Elements im Block aus (4). Diese relative Adresse sei $b_{i,j}$.
Es befinden sich dann

$$s := \min(N_S - i + 1, N_Z - j + 1) \tag{7}$$

Diagonalelemente in diesem Block. Die relativen Adressen der weiteren Diagonalelemente in diesem Block ergeben sich aus der ersten durch Inkrementieren mit $N_S + 1$. Enthält die Diagonale mehr als s Elemente, so ist der zweite Diagonalenabschnitt in dem Block gespeichert, der das Element $d_{r+s,t+s}$ enthält; analog berechnen sich die Adressen der weiteren Diagonalelemente.

Aus dem obengenannten ergibt sich folgender Zusammenhang für den Adressierungsaufwand beim Zugriff auf die veschiedenen Matrixteilstrukturen:

Den geringsten Adressierungsaufwand verursacht i. allg. der Zugriff auf einen Block. Je $N_S N_Z$ Elemente ist einmal Formel (3) auszuwerten. Der Adressierungsaufwand W je Element beträgt demnach

$$W = \frac{K_3}{N_S N_Z}, \quad K_3 : \text{Aufwand zum Auswerten von Formel (3).} \tag{8}$$

Für die Adressierung von Zeilen und Spalten ist einmal Formel (3) und einmal Formel (4) auszuwerten; die weiteren Blockadressen ergeben sich durch Inkrementieren. Die relativen Adressen innerhalb des Blockes sind für alle Blöcke gleich. Der Adressierungsaufwand W pro Element beträgt demnach für Zeilen

$$W = \frac{K_3 + K_4}{N_2}, \quad K_4 : \text{Aufwand zum Auswerten von Formel (4),} \tag{9}$$

bzw. für Spalten

$$W = \frac{K_3 + K_4}{N_1}. \tag{10}$$

Der Adressierungsaufwand für Diagonalen bzw. Nebendiagonalen hängt von der speziellen Wahl der Diagonale ab. Pro angesprochenem Block sind Formel (3) und (4) je einmal auszuwerten. Pro Block können 1 bis maximal $s = \min(N_S, N_Z)$ Diagonalelemente enthalten sein. Für den Adressierungsaufwand W pro Element gilt damit

$$K_3 + K_4 \leq W \leq \frac{K_3 + K_4}{\min(N_S, N_Z)}. \tag{11}$$

Für den Zugriff auf einzelne Matrixelemente gilt stets

$$W = K_3 + K_4.$$

Effizienzuntersuchungen

Einfluß der Blockzerlegung auf die Effizienz

Die Effizienz des Datenzugriffs wird wesentlich durch die Wahl der Blockzerlegung und die Größe des Datenpuffers, d.h. des hauptspeicherresidenten Teils der Matrix, bestimmt. Die Größen N_S und N_Z geben die Größe und Gestalt der Matrixblöcke an; die Größe M_K bestimmt, wieviele dieser Blöcke gleichzeitig im Speicher liegen können.

Bei einem verhältnismäßig regelmäßigen Zugriff auf die Matrix, z.B. Zeile für Zeile oder Spalte für Spalte, wie er für die meisten Matrixalgorithmen typisch ist, ist es aus mehreren Gründen günstig, die Blöcke und den Datenpuffer möglichst groß zu wählen:

1. Pro Matrixzeile bzw. -spalte oder -diagonale werden nur wenige Blöcke benötigt; dies senkt den Aufwand für Adreßrechnung und Pufferverwaltung.

2. Da die Blockdimensionen groß sind, treten beim Kopieren der Daten vom Datenpuffer in den Ausgabevektor des Anwenderprogramms (oder umgekehrt) große Vektorlängen auf. Dies steigert die Effizienz des Kopiervorgangs.

3. Pro I/O-Request werden viele Daten übertragen (große Recordlängen); dies senkt den I/O-Overhead, der pro Datenwort anfällt (höhere Datentransferraten).

Bei einem sehr zufälligen Zugriff auf die Daten kann es allerding sinnvoll sein, die Blöcke nicht zu groß zu wählen, um nicht zu viele nicht benötigte Daten in den Datenpuffer zu laden.

Die optimale Wahl der Werte N_S , N_Z und M_K hängt von mehreren Bedingungen ab; zum einen davon, welche Größen optimiert werden sollen (z.B. CPU-Zeit, Anzahl der I/O-Requests, Anzahl der übertragenen Datenworte oder Speicherplatzbedarf usw.); zum anderen von dem zu realisierenden Matrixalgorithmus und dem daraus resultierenden Zugriffsmuster.

Für einfache Zugriffsmuster können optimale Werte für N_S , N_Z und M_K leicht angegeben werden. Bei komplexen Algorithmen wird ihre Bestimmung i. allg. nur heuristisch möglich sein.

Bei einem rein sequentiellen Zugriff z.B. auf Zeilen der Matrix ist es offenbar optimal, als Blöcke mehrere Matrixzeilen zusammenzufassen ($N_Z = N_2$) und nur einen Block im Puffer zu halten ($M_K = 1$). N_S sollte so groß gewählt werden wie es der verfügbare Speicherplatz erlaubt. Auf diese Weise werden mit jeder I/O-Operation N_S Zeilen transferiert, so daß pro Zugriff auf eine Matrixzeile $1/N_S$ I/O-Operationen entfallen. Die Vektorlänge beim Umkopieren ist $VL = N_Z = N_2$ (da die ganze Zeile in einem Block enthalten ist) und damit maximal. Die Recordlänge der I/O-Operationen ist $N_S N_Z$, was wegen $M_K = 1$ der Größe des gesamten Datenpuffers entspricht. Bei fest vorgegebener Puffergröße ist sie damit maximal. Die Pufferausnutzung ist ebenfalls optimal, da nur Daten transferiert werden, die auch benötigt werden.

Wird auf Zeilen **und** Spalten (z.B. alternierend) zugegriffen, so optimiert die obige Wahl von N_S , N_Z und M_K zwar den Zugriff auf Zeilen, bewirkt jedoch ein sehr ungünstiges Verhalten beim Zugriff auf Spalten, da die ganze Matrix beim Zugriff auf eine Spalte transferiert wird.

Geht man zu einer allgemeinen Blockzerlegung mit Blöcken der Größe $N_S \times N_Z$ über, so werden $M_S = N_1/N_S$ Blöcke beim Zugriff auf Spalten und $M_Z = N_2/N_Z$ Blöcke beim Zugriff auf Zeilen benötigt. Da ein Block stets einer Zeile und einer Spalte gemeinsam angehört, müssen sich mindestens

$$M_K = M_S + M_Z - 1 = N_1/N_S + N_2/N_Z - 1 \tag{12}$$

Blöcke im Puffer befinden, wenn die Blöcke für eine Zeile und eine Spalte gepuffert werden sollen. Der Speicherplatzbedarf für den Datenpuffer beträgt dann

$$S(N_S, N_Z) = N_1 N_Z + N_2 N_S - N_S N_Z \geq N_S N_Z. \tag{13}$$

Soll der Speicherplatzbedarf minimiert werden, so wären also N_S und N_Z möglichst klein zu wählen; andererseits bestimmt eine vorgegebene Speichergröße die maximal mögliche Blockgröße.

Geht man von einer idealisierten Matrixdurchlaufung aus, so können nach dem Laden der Blöcke für eine Zeile bzw. eine Spalte N_S weitere Zeilen bzw. N_Z weitere Spalten ohne neue I/O-Operationen angesprochen werden. Bei Vernachlässigung der Tatsache, daß ein Block Zeilen und Spalten gemeinsam ist, beträgt die Anzahl der I/O-Operationen dann pro Zeile

$$R_Z = \frac{N_2}{N_S N_Z}, \tag{14}$$

bzw. pro Spalte

$$R_S = \frac{N_1}{N_S N_Z}. \tag{15}$$

Zur Minimierung der I/O-Operationen wäre also das Produkt $N_S N_Z$ möglichst groß zu wählen. Ist der maximal für den Datenpuffer verfügbare Speicherplatz fest vorgegeben, so ist wegen (13) das Produkt $N_S N_Z$ nach oben beschränkt.

Sei also

$$Q := N_S N_Z$$

fest. Es sind Werte N_S, N_Z zu finden, die (13) minimieren. Mit $N_S = Q/N_Z$ lautet (13)

$$S(N_Z) = N_1 N_Z + \frac{N_2 Q}{N_Z} - Q. \tag{16}$$

(16) ist minimal für

$$N_Z^2 = \frac{N_2 Q}{N_1}$$

bzw. für

$$\frac{N_Z}{N_S} = \frac{N_2}{N_1}.$$

Das heißt, eine Blockzerlegung, die die Anzahl der I/O-Operationen bei vorgegebenem Speicherplatzbedarf minimiert, ist hier gegeben, wenn die Matrix in genausoviele Blöcke pro Zeile wie Blöcke pro Spalte aufgeteilt wird (bei quadratischen Matrizen also in quadratische Blöcke).

An diesem Sachverhalt ändert sich auch nichts, wenn z.B. häufiger auf Zeilen als auf Spalten zugegriffen wird, da mit (14) auch zugleich (15) minimiert wird. Erst wenn man die Voraussetzung (12) fallenläßt und z.B. nur noch die Blöcke für den Zeilenzugriff im Datenpuffer hält, kann eine neue Blockzerlegung gewählt werden, um den Zugriff auf Zeilen zu optimieren.

Es soll nun ermittelt werden, wann eine Pufferung von Zeilen und Spalten nicht mehr lohnt, sondern vielmehr eine Pufferung von nur Zeilen bei Verwendung "horizontaler" Blöcke ($N_S << N_Z$) sinnvoll ist. Dabei soll zunächst nur die Häufigkeit von I/O-Operationen als alleiniges Kriterium betrachtet werden.

Sei k die Anzahl von Zeilenzugriffen, die je Spaltenzugriff erfolgen. Wie in (14) soll die idealisierte Annahme gemacht werden, daß eine vollständige Pufferausnutzung beim Zeilenzugriff gegeben sei. Für den vorliegenden Fall bedeutet dies, daß die relativ seltenen Spaltenzugriffe nur erfolgen sollen, wenn die Daten des "Zeilenpuffers" nicht mehr benötigt werden.

Es sei $N_S \times N_Z$ die bei der Pufferung von Zeilen und Spalten verwendete Blockgröße (Methode A), $N_S' \times N_Z'$ ($N_Z' \geq N_Z$) die Blockgröße bei Pufferung von nur Zeilen (Methode B). Die Anzahl der I/O-Operationen beim Zugriff auf k Zeilen und eine Spalte beträgt dann gemäß (14) und (15) bei Methode A

$$I_a = k\frac{N_2}{N_S N_Z} + \frac{N_1}{N_S N_Z} \tag{17}$$

und bei Methode B

$$I_b = k\frac{N_2}{N_S'N_Z'} + \frac{N_1}{N_S'}. \tag{18}$$

Der Schnittpunkt von (17) und (18) ergibt sich zu

$$k_0 = \frac{N_Z'N_1(N_SN_Z - N_S')}{N_2(N_S'N_Z' - N_SN_Z)}. \tag{19}$$

Betrachtet man speziell quadratische Matrizen ($N_1 = N_2$) und geht davon aus, daß der bei Methode A für die Pufferung von Spalten verbrauchte Speicherplatz jetzt für die Zeilenpufferung verwendet wird ($N_S' = 2N_S$), so geht (19) über in

$$k_0 = \frac{N_Z'(N_Z - 2)}{2N_Z' - N_Z}. \tag{20}$$

Wegen der speziellen Voraussetzungen hängt (20) von der Matrixgröße und der Spaltendimension der Blöcke nicht mehr ab.

Für das Beispiel einer quadratischen Matrix mit $N_1 = N_2 \geq 240$ und einer Blockzerlegung mit $N_Z = 20$ nach Methode A sowie einer Blockzerlegung in "horizontale" Blöcke mit $N_Z' = 240$ nach Methode B liefert (20) den Wert $k_0 = 9.4$. D.h. eine Aufgabe der Pufferung von Zeilen und Spalten bei einer Blockzerlegung in quadratische Blöcke nach Methode A zugunsten der Pufferung nach Methode B lohnt sich, wenn wenigstens 10-mal häufiger auf Zeilen als auf Spalten zugegriffen wird (vgl. Tabelle 2).

Alle obigen Überlegungen gelten analog für den häufigen Zugriff auf Spalten, wenn N_S mit N_Z und N_S' mit N_Z' vertauscht werden.

Tabelle 1 zeigt einige aus (20) gewonnene Werte für k_0 bei festem Wert $N_Z = 100$.

N_Z	N_Z'	k_0
100	1000	51.58
100	500	54.44
100	250	61.25
100	200	65.33
100	110	89.83
100	100	98.00

Tabelle 1: Wertetabelle zu Formel (20).

Für $N_Z = N_Z' = 100$ liegt der Fall vor, daß bei Methode B keine horizontalen Blöcke für die Optimierung des Zeilenzugriffs verwendet , sondern vielmehr die Zeilendimension von Methode A (bei Verdoppelung der Spaltendimension) beibehalten wurde. Dies hat zur Folge, daß beim Spaltenzugriff bei Methode B nicht mehr Daten transferiert werden als auch bei Methode A. Da für $k_0 \geq 98$ (bzw. allgemein für $k_0 \geq N_Z - 2$) gleichzeitig die Anzahl der I/O-Operationen sinkt, ist diese Blockzerlegung also sowohl hinsichtlich der Anzahl der I/O-Requests als auch bezüglich der beim I/O übertragenen Datenmenge günstiger als Methode A. Diese letzte Aussage hängt allerdings in der Praxis stark davon ab, daß tatsächlich der ganze "Zeilenpuffer", der bei Methode B doppelt so groß ist wie bei Methode A, störungsfrei ausgelesen werden kann.

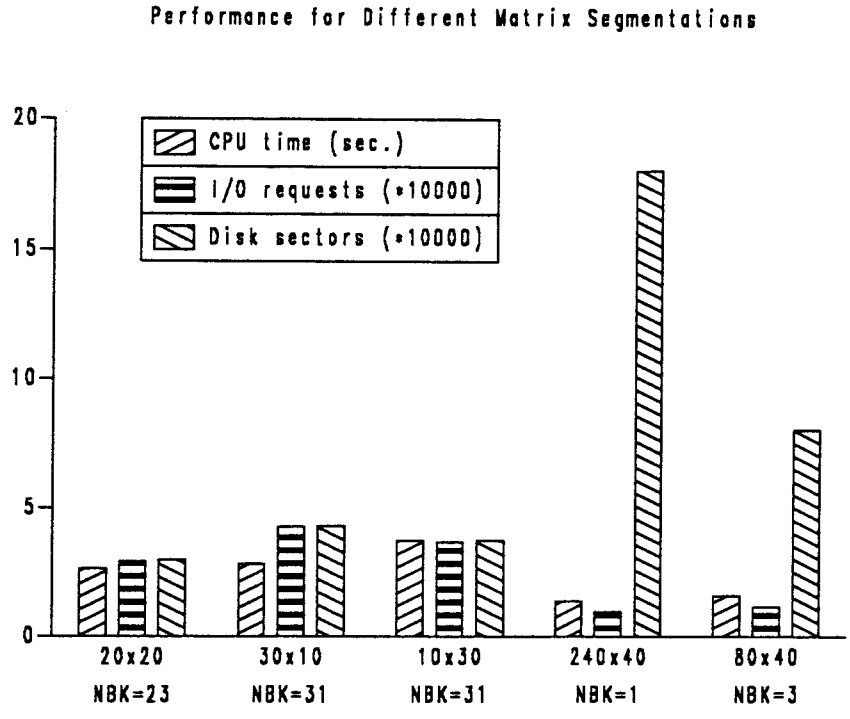


Abbildung 3. Paging-Performance in Abhängigkeit von der Blockzerlegung

Abbildung 3 auf Seite 10 und Tabelle 2 zeigen in Ergänzung zu den obigen Ergebnissen den Einfluß der Blockzerlegung auf die CPU-Zeit, die Anzahl der I/O-Requests und die Anzahl der übertragenen Datenworte bei annähernd gleichem Speicherplatzbedarf am Beispiel des LINPAK-Programms SGEFA/SGESL zur Lösung linearer Gleichungssysteme für eine 240 × 240 -Matrix. Bei diesem Algorithmus wird ca. 50-mal häufiger auf Spalten (bzw. Teile davon) als auf Zeilen zugegriffen. Die in Tabelle 2 wiedergegebenen Daten sind Meßwerte, die bei Läufen auf einer CRAY X-MP gewonnen wurden. Als Pufferverwaltungsstrategie wurde FIFO verwendet.

Nr.	N_S	N_Z	M_K	Puffer-größe	CPU	I/O-Requ.	Disk-Sect.	I/O-Wait
1	20	20	23	9200	2.636	29256	29729	15.7
2	30	10	31	9300	2.819	42615	43090	19.0
3	10	30	31	9300	3.743	36775	37248	17.0
4	240	40	1	9600	1.372	9569	180486	23.0
5	80	40	3	9600	1.613	11534	80620	13.7

Tabelle 2: Meßwerte zu Abbildung 3.

Es bestätigt sich, daß die für den einfachen Fall der Durchlaufung in Zeilen und Spalten gefundene optimale Blockzerlegung auch hier optimal ist (Lauf Nr. 1), solange die Voraussetzung gilt, daß der Puffer sowohl Blöcke für eine Zeile als auch für eine Spalte aufnehmen soll. Obwohl sehr viel häufiger auf Spalten als auf Zeilen zugegriffen wird, ist die Zerlegung in "spaltenorientierte" rechteckige Blöcke (Lauf Nr. 2) keineswegs günstiger.

Lauf Nr. 4 zeigt die Blockoptimierung nur bezüglich des Spaltenzugriffs (jeder Block besteht aus 40 Spalten der Matrix). Das hat zur Folge, daß die CPU-Zeit und die Anzahl der I/O-Requests um den Faktor 2 bzw. 3 sinken; dafür steigt die Anzahl der übertragenen Disk-Sektoren um den Faktor 6 an. Lauf Nr. 5 stellt einen Kompromiß zwischen Lauf 1 und Lauf 4 dar. Wie in Lauf 4 werden nur die Blöcke für den Spaltenzugriff gleichzeitig im Puffer gehalten. Da die Blöcke um den Faktor 3 kleiner sind, wird beim Zugriff auf Zeilen allerdings höchstens ein Drittel der Matrix referiert. Der CPU-Verbrauch und die Anzahl der I/O-Requests ist damit um ca. 20 Prozent größer als bei Lauf 4, die transferierte Datenmenge aber um den Faktor 2.2 kleiner.

Abbildung 4 auf Seite 11 zeigt die CPU-Zeit und das I/O-Verhalten für das gleiche Programm in Abhängigkeit von der Anzahl der im Puffer gespeicherten Blöcke. Für den gezeigten Programmlauf ist $N_1 = N_2 = 240$, es liegt eine Blockzerlegung in quadratische Blöcke der festen Größe $N_S \times N_Z = 20 \times 20$ vor; die Anzahl der gepufferten Blöcke wurde in mehreren Läufen von $M_K = 144$ bis $M_K = 5$ verringert. Die zugehörigen Meßwerte finden sich in Tabelle 3.

M_K	Puffer-größe	CPU	I/O-Requ.	I/O-Wait
144	57600	1.89	387	2.86
100	40000	2.15	10820	6.61
75	30000	2.34	18444	9.83
50	20000	2.47	21808	11.36
45	18000	2.52	23922	11.12
40	16000	2.54	24358	10.12
35	14000	2.57	25567	11.03
30	12000	2.62	27275	12.57
25	10000	2.62	28067	13.95
20	8000	2.71	31195	13.47
15	6000	2.76	33575	14.32
13	5200	2.77	34026	14.51
12	4800	2.80	34714	14.78
11	4400	7.06	193365	1:12.00
10	4000	10.71	329003	2:08.60
7	2800	18.14	601174	3:47.61
5	2000	20.48	694019	4:26.53

Tabelle 3: Meßwerte zu Abbildung 4.

Es zeigt sich, daß CPU-Zeit und Anzahl der I/O-Requests nur mäßig ansteigen, solange mindestens Blöcke für den Zugriff auf eine Zeile oder eine Spalte im Puffer gespeichert werden können ($M_K \geq 12$). Bei Unterschreiten dieser Grenze allerdings steigen CPU-Zeit und Anzahl der I/O-Requests drastisch an. Die Pufferung von Zeilen **und** Spalten bringt hier also wegen des seltenen Zugriffs auf Zeilen keinen Gewinn.

Abbildung 5 auf Seite 12 bzw. Tabelle 4 zeigen den Einfluß der Blockgröße auf die CPU-Zeit und die Anzahl der I/O-Requests. Die Abbildung gibt die Daten für mehrere Programmläufe für das gleiche Programm mit Matrixdimension 240 × 240 wieder. Es wurde bei allen Läufen eine Zerlegung in quadratische Blöcke gewählt, beginnend mit der maximalen Blockgröße $N_S \times N_Z = 240 \times 240$ bis zu Blöcken der Größe 8 × 8, wobei jeweils die für den Zugriff auf eine Zeile oder Spalte benötigten Blöcke gepuffert wurden. Bis zu einer Blockgröße $\geq 48 \times 48$ ändert

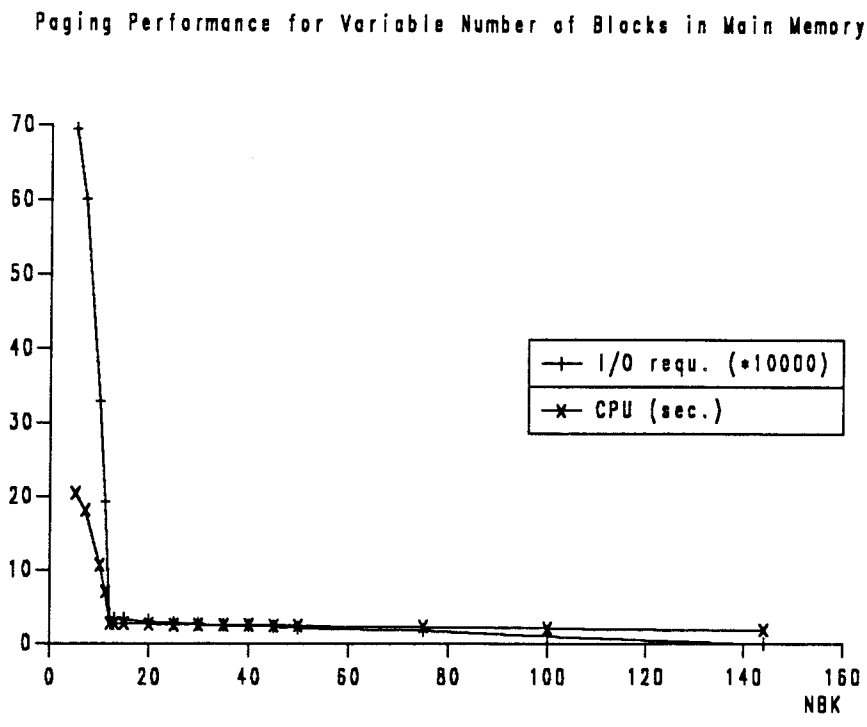


Abbildung 4. Paging-Performance in Abhängigkeit von der Blockanzahl

sich der CPU-Zeitverbrauch nur mäßig (Steigerung um den Faktor 1.5), und auch die Anzahl der I/O-Requests bleibt in einer vertretbaren Größenordnung. Bei Blockgrößen $\leq 40 \times 40$ steigt dann aber die Anzahl der I/O-Requests und bei $N_S \times N_Z \leq 20 \times 20$ auch die CPU-Zeit stark an.

Zum Vergleich ist der Speicherverbrauch für den Datenpuffer in Prozent von der Gesamtmatrix in der Abbildung mit aufgetragen. Je nach Gewichtung von CPU-Zeit, I/O-Requests und Speicherbedarf dürfte eine optimale Blockzerlegung für diese Matrixgröße unter der Voraussetzung quadratischer Blöcke im Bereich $20 \times 20 \leq N_S \times N_Z \leq 60 \times 60$ liegen.

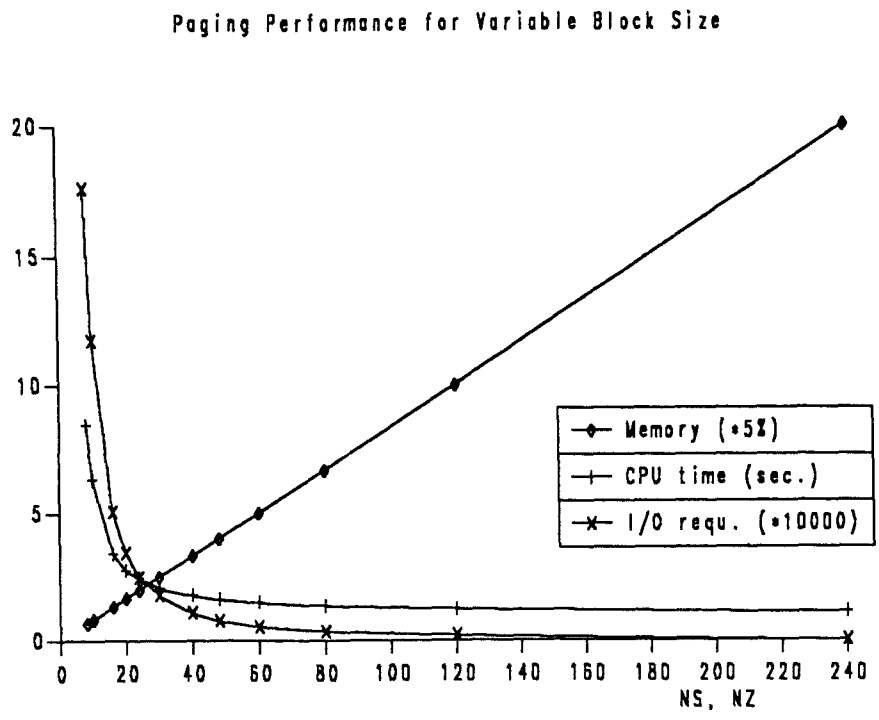


Abbildung 5. Paging-Performance in Abhängigkeit von der Blockgröße

$N_S \times N_Z$	M_K	Puffer-größe	CPU	I/O-Requ.	I/O-Wait
240 × 240	1	57600	1.11	103	2.6
120 × 120	2	28800	1.24	2247	11.3
80 × 80	3	19200	1.37	3735	14.2
60 × 60	4	14400	1.52	5622	11.4
48 × 48	5	11520	1.64	7957	11.0
40 × 40	6	9600	1.79	11144	12.5
30 × 30	8	7200	2.08	17907	14.4
24 × 24	10	5760	2.49	25023	14.8
20 × 20	12	4800	2.77	34713	17.2
16 × 16	15	3840	3.41	50735	24.6
10 × 10	24	2400	6.34	117535	8:38.3
8 × 8	30	1920	8.46	176236	43:52.3

Tabelle 4: Meßwerte zu Abbildung 5.

Vergleich zwischen FIFO- und LRU-Pufferverwaltung

Zur Verwaltung des Datenpuffers wurden zwei verschiedene Strategien untersucht, "First In First Out" (FIFO) und "Least Recently Used" (LRU).

FIFO: Die Blöcke des Puffers werden zyklisch vergeben, beginnend mit dem ersten Block des Puffers. Ist der Puffer voll, so wird der Block verdrängt, der sich am längsten im Puffer befindet.

LRU: Bei jedem Zugriff auf einen Block des Puffers wird der Zeitpunkt des Zugriffs gespeichert. Ist der Puffer voll, so wird der Block verdrängt, auf den am längsten nicht mehr zugegriffen wurde.

FIFO hat den Vorteil, keinen besonderen Verwaltungsaufwand zu verursachen. Damit ist die Implementierung einfach und das Blockladen sehr schnell. Nachteilig ist, daß beim Verdrängen eines Blockes kein Bezug zum Zugriffsverhalten des Programmes genommen wird.

LRU hat sich bei Paging-Systemen dort als geeignet erwiesen, wo das Zugriffsverhalten gewisse "Lokalitätseigenschaften" zeigt (d.h. aus einem großen Adreßbereich wird ein kleiner Teil bevorzugt angesprochen).[1] Nachteilig bei LRU ist, daß beim Laden eines Blocks zunächst durch einen Suchvorgang festgestellt werden muß, welcher Block zu verdrängen ist. Diese Vorgehensweise ist besonders bei Software-Implementierungen recht aufwendig. Bei Matrixalgorithmen ist ein lokales Zugriffsverhalten beim Datenzugriff nicht unbedingt zu erwarten, da oft die Matrizen mehr oder weniger regelmäßig vollständig durchlaufen werden. Damit ist das Zugriffsverhalten zu Programmdaten sicherlich wesentlich verschieden vom Zugriff eines Programms auf den Programmcode (z.B. Schleifen o.ä.).

Zum Vergleich von FIFO und LRU sollen zwei Anwendungsbeispiele genauer betrachtet werden.

Beispiel 1: Durchlaufung einer Matrix in Zeilen und Spalten.

Eine quadratische $N \times N$-Matrix werde "sequentiell" in Zeilen und Spalten durchlaufen (d.h. Zeile 1, Spalte 1, Zeile 2, Spalte 2, usw.). Es liege eine Zerlegung der Matrix in quadratische Blöcke vor, wobei Pufferplatz für die Blöcke einer Zeile und einer Spalte zur Verfügung stehe ($M_K = M_S + M_Z - 1$). Der Puffer sei bereits mit den Blöcken einer Zeile i und einer Spalte j gefüllt. Beim nachfolgenden Zugriff auf Zeile i + 1 und Spalte j + 1 soll die Situation vorliegen, daß Blöcke nachgeladen werden müssen (d.h. Zeile i und i + 1 sowie Spalte j und j + 1 liegen nicht in denselben Blöcken). Seien $B_{i,k}$, $1 \le k \le M_Z$ die zu Zeile i und $B_{n,j}$, $1 \le n \le M_S$ die zu Spalte j gehörigen Blöcke.

Das Nachladen der Blöcke verläuft folgendermaßen:

FIFO:

$B_{i+1,1}$ bis $B_{i+1,j-1}$ laden,
$B_{i+1,j}$ wiederbenutzen,
$B_{i+1,j+1}$ bis B_{i+1,M_Z} laden.
(Block $B_{i,j+1}$ wird überschrieben)

Beim Zeilenzugriff treten also $M_Z - 1$ Blockloads auf.

$B_{1,j+1}$ bis $B_{i,j+1}$ laden,
$B_{i+1,j+1}$ wiederbenutzen,
$B_{i+2,j+1}$ bis $B_{M_S,j+1}$ laden.
(Block $B_{i+1,j}$ wird überschrieben)

Beim Spaltenzugriff treten damit $M_S - 1$ Blockloads auf.

---

[1] Vgl. z.B.: Peter J. Denning: "On modeling program behavior"; Spring Joint Computer Conference, 1972.

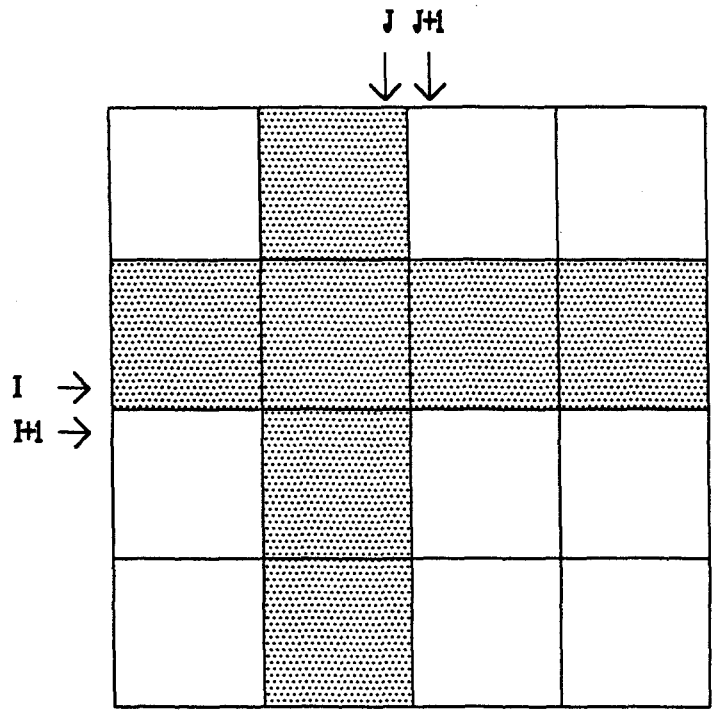


Abbildung 6. Blockwechsel beim Zugriff auf Zeilen und Spalten

Da $B_{i+1,j}$ beim Spaltenzugriff auf Spalte $j + 1$ wieder überschrieben wird, muß dieser Block beim nächsten Zeilenzugriff (auf Zeile $i + 2$) wieder geladen werden.

Insgesamt sind also je N_Z Zeilenzugriffe und N_S Spaltenzugriffe

$$M_S + M_Z - 1 = 2M_S - 1$$

Blockloads erforderlich.

LRU:

$B_{i+1,1}$ bis $B_{i+1,j-1}$ laden,
$B_{i+1,j}$ wiederbenutzen,
$B_{i+1,j+1}$ bis B_{i+1,M_Z} laden.
(Block $B_{i,j+1}$ wird überschrieben, da lange nicht mehr angesprochen).

Beim Zeilenzugriff treten damit $M_Z - 1$ Blockloads auf.

$B_{1,j+1}$ bis $B_{i,j+1}$ laden,
$B_{i+1,j+1}$ wiederbenutzen,
$B_{i+2,j+1}$ bis $B_{M_S,j+1}$ laden.
(Block $B_{i+1,j}$ wird nicht überschrieben, da kürzlich referiert).

Beim Spaltenzugriff treten hier $M_S - 1$ Blockloads auf.

Insgesamt sind also je N_Z Zeilenzugriffe und N_S Spaltenzugriffe

$$M_S + M_Z - 2 = 2M_S - 2$$

Blockloads erforderlich.

Der Aufwand beim vollständigen Durchlaufen der Matrix in Zeilen und Spalten beträgt damit bei FIFO

$$M_S(2M_S - 1)$$

und bei LRU

$$M_S(2M_S - 2) + 1.$$

Die relative Ersparnis FIFO/LRU beträgt also

$$\frac{M_S(2M_S - 1)}{M_S(2M_S - 2) + 1} \underset{M_S \to \infty}{\to} 1,$$

verschwindet also mit wachsender Blockanzahl.

Beispiel 2: Durchlaufung einer Matrix in Diagonalen.

Es sei eine quadratische $N \times N$ -Matrix mit Blockzerlegung in quadratische Blöcke der Größe $N_S \times N_Z$ gegeben. Die Puffergröße sei $M_K = M_S + M_Z - 1 = 2M_S - 1$.

1. Fall: Durchlaufen der oberen Dreiecksmatrix in Diagonalen, beginnend mit der Hauptdiagonalen.

FIFO:

1. Beim Zugriff auf die Hauptdiagonale werden die Diagonalblöcke ($B_{i,i}$, $i = 1,2, \ldots, M_S$) in aufeinanderfolgenden Pufferblöcken gespeichert.

2. Beim Zugriff auf die 1. obere Nebendiagonale werden die zusätzlich benötigten Blöcke $B_{i,i+1}$, $i = 1,2, \ldots, M_S - 1$ in den darauffolgenden Pufferpositionen gespeichert (kein Verdrängen noch benötigter Blöcke!).

3. Beim Zugriff auf die $N_Z + 1$-te obere Nebendiagonale werden die Diagonalblöcke $B_{i,i}$ obsolet, sie werden wegen $M_K = 2M_S - 1$ durch die neu benötigten Blöcke $B_{i,i+2}$, $i = 1,2, \ldots, M_S - 2$ überschrieben (kein Verdrängen noch benötigter Blöcke!).

Wesentlich für das optimale I/O-Verhalten von FIFO ist, daß Diagonalblöcke und Nebendiagonalblöcke im Puffer jeweils in zusammenhängenden Blöcken gespeichert und nicht etwa miteinander vermischt sind. Dadurch tritt keine Verdrängung noch benötigter Blöcke auf (vgl. Fall 2).

LRU: Das LRU-I/O-Verhalten für dieses Beispiel ist im Ergebnis das gleiche wie bei FIFO.

2. Fall: Durchlaufen der oberen Dreiecksmatrix in Diagonalen, beginnend mit der 1. oberen Nebendiagonale.

FIFO:

1. Beim Zugriff auf die 1. obere Nebendiagonale werden die Blöcke $B_{1,1}$, $B_{1,2}$, $B_{2,2}$, $B_{2,3}$, ... im Puffer gespeichert, d.h. Hauptdiagonalblöcke und Nebendiagonalblöcke sind miteinander vermischt. Nach Einlesen aller benötigten Blöcke ist der Puffer voll.

2. Beim Zugriff auf die N_Z-te Nebendiagonale werden nur Blöcke angesprochen, die bereits geladen sind.

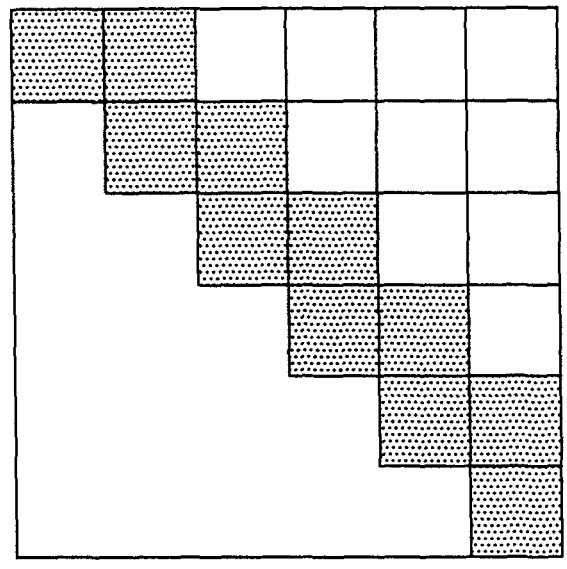


Abbildung 7. Blockwechsel beim Zugriff auf Diagonalen

3. Beim Zugriff auf die N_Z + 1-te obere Nebendiagonale überschreiben die neu benötigten Blöcke $B_{i,i+2}$, $i = 1,2, \ldots, M_S - 2$ beim Laden $M_S/2 - 1$ Stück der nicht mehr benötigten Diagonalblöcke sowie $M_S/2 - 1$ Stück der noch benötigten Blöcke $B_{i,i+1}$. Da die $B_{i,i+1}$ vor den $B_{i,i+2}$ angesprochen werden, können die $B_{i,i+1}$ jeweils noch einmal wiederbenutzt werden, bevor sie überschrieben werden. Die Folge davon ist, daß die neu eingelesenen Nebendiagonalblöcke $B_{i,i+2}$ in aufeinanderfolgenden Positionen im Puffer liegen.

4. Beim Zugriff auf die $N_Z + 2$-te obere Nebendiagonale werden die gerade überschriebenen Blöcke $B_{i,i+1}$ wieder eingelesen; sie werden in aufeinanderfolgenden Positionen im Puffer gespeichert; hierbei werden noch einmal Blöcke überschrieben, die in folgenden Aufrufen benötigt werden.

Nach wenigen vollständigen Blockladezyklen für den Zugriff auf die ersten Nebendiagonalen hat sich der Puffer "synchronisiert"; d.h. die vermischte Speicherung von neu geladenen und nicht mehr benötigten Blöcken ist aufgehoben. Es tritt keine weitere Verdrängung von noch benötigten Blöcken auf. Der Synchronisationseffekt kann beschleunigt werden, wenn die Anzahl der im Puffer befindlichen Blöcke vergrößert wird (vgl. Tabelle 5).

LRU:

1. Der Zugriff auf die 1. obere Nebendiagonale bewirkt die gleiche Pufferbelegung wie bei FIFO.

2. Beim Zugriff auf die N_Z-te obere Nebendiagonale werden nur die schon geladenen Nebendiagonalblöcke $B_{i,i+1}$ angesprochen. Ihr Alter wird aktualisiert; damit werden die nicht mehr benötigten Diagonalblöcke $B_{i,i}$ Kandidaten für zu verdrängende Blöcke.

3. Beim Zugriff auf die N_Z + 1-te obere Nebendiagonale verdrängen die neu benötigten Blöcke $B_{i,i+2}$, $i = 1,2, \ldots, M_S - 2$ nur nicht mehr benötigte Diagonalblöcke $B_{i,i}$.

4. Beim Zugriff auf die N_Z + 2-te obere Nebendiagonale werden nur bereits geladene Blöcke angesprochen.

Bei LRU tritt keine Verdrängung noch benötigter Blöcke auf, da diese Blöcke stets nach den nicht mehr benötigten referiert werden, wodurch ihr Alter aktualisiert wird.

Tabelle 5 zeigt das I/O-Verhalten von FIFO und LRU beim Zeilen/Spalten-Zugriff sowie beim Zugriff auf Diagonalen noch einmal an Hand der Meßwerte für einige Programmbeispiele. Allen Läufen, mit Ausnahme von Lauf Nr. 1, liegt eine 200 × 200 -Matrix zugrunde (Lauf 1, 100 × 100-Matrix). Für die Blockzerlegung wurden in allen Fällen quadratische Blöcke der Größe 20 × 20 gewählt. Damit gilt stets $M_S = M_Z = 10$ (bzw. $M_S = M_Z = 5$ für Lauf 1).

Die Läufe 1 und 2 zeigen das I/O-Verhalten bei alternierendem Zugriff auf Zeilen und Spalten der Matrix für Matrixgröße 100 × 100 (Lauf 1) bzw. 200 × 200 (Lauf 2). Die im vorigen Abschnitt gefundene Ersparnis von $M_S - 1$ Blockloads für eine vollständige Matrixdurchlaufung bestätigt sich exakt. Damit beträgt der relative Gewinn von LRU gegenüber FIFO ca. 10 Prozent für Lauf 1 bzw. 5 Prozent für Lauf 2.

Lauf 3 zeigt die Durchlaufung der oberen Dreiecksmatrix einer 200 × 200 -Matrix in Diagonalen, beginnend mit der Hauptdiagonalen. Es bestätigt sich, daß FIFO und LRU hier gleichwertig sind. Bei beiden werden 55 Blöcke geladen, also genau die Anzahl der in der oberen Dreiecksmatrix enthaltenen Blöcke. Es findet also kein unnötiges Blockladen statt.

Lauf Nr.	M_K	LRU/ FIFO	Block-loads	CPU	Bemerkungen
1	9	FIFO	45	0.0090	Zeilen/Spalten-Zugriff
1	9	LRU	41	0.0096	
2	19	FIFO	190	0.0206	Wie Nr. 1, aber Matrix-
2	19	LRU	181	0.0232	größe 200x200
3	19	FIFO	55	0.0243	Diagonalendurchlaufung
3	19	LRU	55	0.0256	mit Hauptdiagonale
4	19	FIFO	67	0.0202	N/2 Diagonalen ohne
4	19	LRU	45	0.0206	Hauptdiagonale
5	19	FIFO	77	0.0244	N Diagonalen ohne
5	19	LRU	55	0.0249	Hauptdiagonale
6	23	FIFO	58	0.0244	Wie Nr.5, M_K vergrößert
7	19	FIFO	441	0.0402	Diagonalenzugriff mit
7	19	LRU	825	0.0556	"Störung"
8	22	FIFO	63	0.0290	Wie Nr. 7, aber
8	22	LRU	57	0.0298	Puffer vergrößert

Tabelle 5: Meßwerte zum Vergleich zwischen FIFO und LRU.

In den Läufen 4, 5 und 6 wurde die obere Dreiecksmatrix ohne die Hauptdiagonale durchlaufen, beginnend mit der 1. oberen Nebendiagonale. In Lauf 4 wurden zunächst nur die ersten 100 Nebendiagonalen durchlaufen. FIFO benötigt hier 22 Blockloads mehr als LRU, bedingt durch die vermischte Speicherung von "neuen" und "alten" Blöcken im Puffer zu Beginn des Zugriffs. Lauf 5 belegt, daß sich der Puffer nach der Durchlaufung der ersten 100 Diagonalen "synchronisiert" hat, denn beim Durchlaufen der vollen Dreiecksmatrix (ohne Hauptdiagonale) beträgt der Unterschied zwischen FIFO und LRU ebenfalls genau 22 Blockloads. In Lauf 6 wurde der Datenpuffer für FIFO um 4 Blöcke vergrößert. Damit verbessert sich das I/O-Verhalten deutlich gegenüber Lauf 5.

Die Läufe 7 und 8 schließlich zeigen, daß das bisher gefundene I/O-Verhalten von FIFO und LRU stark von dem sehr regelmäßigen Blockzugriff bei der Diagonalendurchlaufung und der korrekten Wahl der Puffergröße abhängen. In Lauf 7 und 8 wurde nach jedem Diagonalenzugriff auf 3 beliebige, aber fest gewählte Matrixelemente zugegriffen, um den regelmäßigen Datenzugriff zu stören. In Lauf 7 wurde zunächst die alte Puffergröße beibehalten. Es zeigt sich, daß sowohl FIFO als auch LRU beim I/O-Verhalten sehr empfindlich reagieren, da der Puffer durch die zusätzlichen Datenzugriffe jetzt zu klein geworden ist. LRU benötigt jetzt fast doppelt soviele Blockloads wie FIFO. Die Vergrößerung des Puffers um die 3 zusätzlich benötigten Blöcke (Lauf 8) stellt die alten Verhältnisse annähernd wieder her. LRU erweist sich trotz "Störung" als sehr stabil. FIFO benötigt immerhin 8 Blockloads mehr als in Lauf 3.

Insgesamt zeigt sich, daß die LRU-Pufferverwaltung speziell beim Diagonalenzugriff deutlich günstiger sein kann als FIFO; allerdings weist LRU in allen Läufen einen höheren CPU-Verbrauch auf als FIFO (selbst dann, wenn bei LRU deutlich weniger I/O durchgeführt wurde).

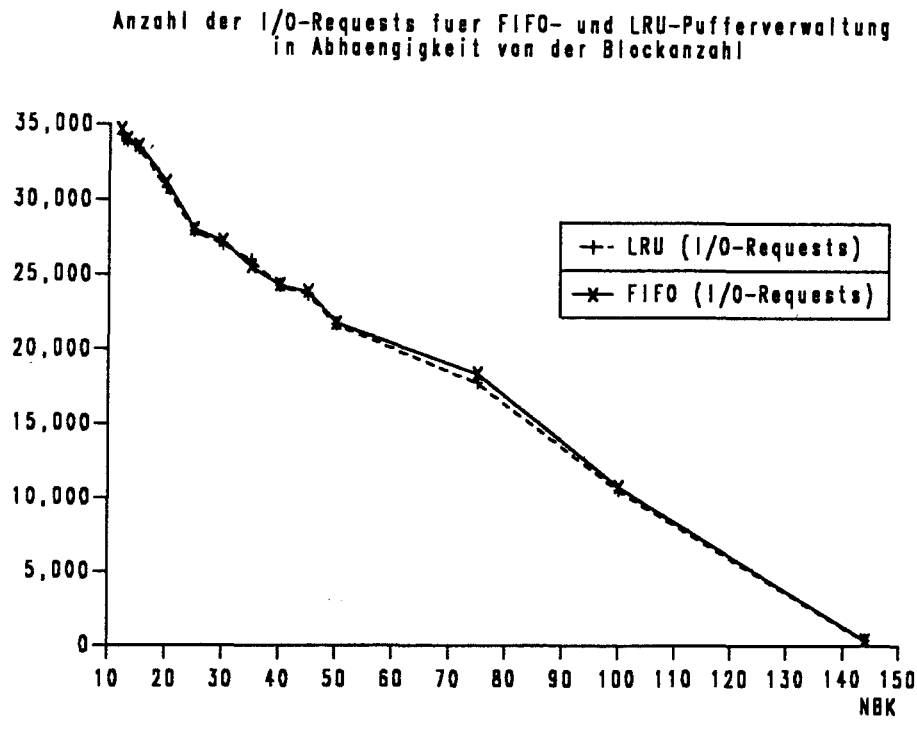


Abbildung 8. Anzahl der I/O-Requests für FIFO und LRU.

Ein abschließender Vergleich soll noch einmal am Beispiel des Programms SGEFA/SGESL gezogen werden (Matrixgröße 240 × 240 , Blockgröße 20 × 20). Wie im Abschnitt "Einfluß der Blockzerlegung auf die Effizienz" auf Seite 6 wurden mehrere Läufe mit unterschiedlicher Puffergröße durchgeführt. Abbildung 8 auf Seite 19 zeigt das I/O-Verhalten, Abbildung 9 auf Seite 20 den CPU-Verbrauch für FIFO und LRU. Während die Anzahl der I/O-Requests bei LRU im günstigsten Fall um 6 Prozent niedriger liegt als bei FIFO (M_K = 75), ist der CPU-Verbrauch in praktisch allen Fällen bei LRU um wenigstens 15 Prozent höher als bei FIFO.

Eine generelle Bevorzugung von LRU gegenüber FIFO als Pufferverwaltungsstrategie scheint damit nicht gerechtfertigt zu sein.[2]

Effizienz der Ein-/Ausgabe-Routinen

Wegen der Blockzerlegung der Matrizen werden bei der vorliegenden Konzeption des Software-Paging-Verfahrens Datenrecords wahlfrei gelesen und geschrieben. Für die Realisierung der Ein-/Ausgabeoperationen kommen daher nur "Direct Access"-Routinen bzw. hiermit funktional gleichwertige Ein-/Ausgabe-Routinen in Betracht.

CRAY-Ein-/Ausgabe-Routinen

Auf der CRAY steht eine Vielzahl von Ein-/Ausgabe-Routinen zur Durchführung von "Direct Access I/O" bzw. dem damit gleichwertigen "Random I/O" zur Verfügung. Um die geeignetste I/O-Realisierung zu ermitteln, wurden Vergleichsläufe für folgende I/O-Methoden durchgeführt:

---

[2] In den vorliegenden CRAY- und IBM-Implementierungen wurde FIFO-Pufferverwaltung verwendet.

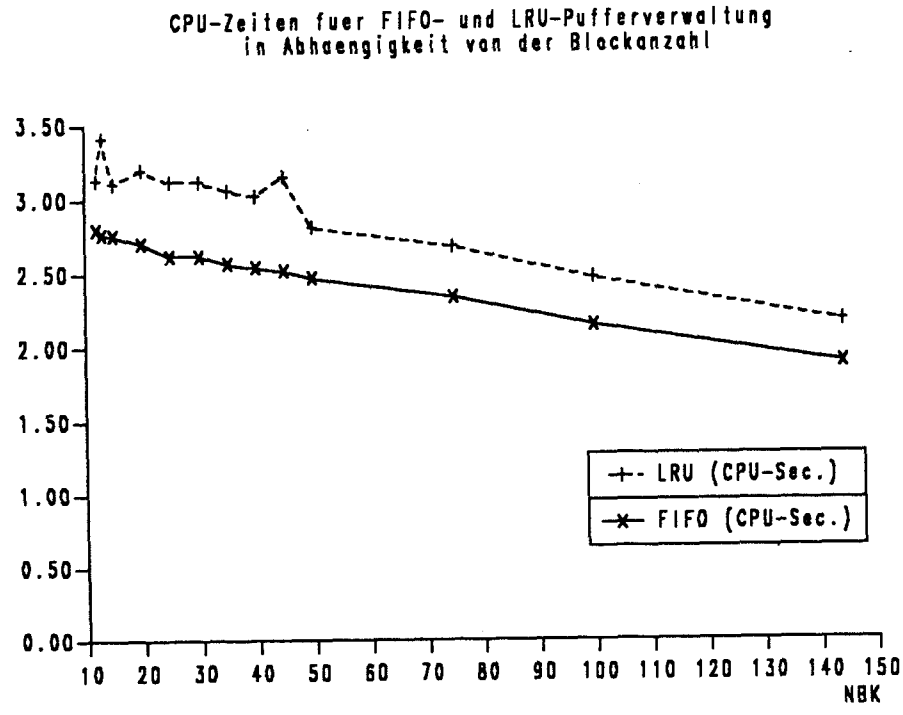


Abbildung 9. CPU-Zeit für FIFO und LRU.

- Standard Direct Access,
- READMS/WRITMS-Routinen (Random I/O, Record-orientiert),
- READDR/WRITDR-Routinen (ungeblockter Random I/O, Record-orientiert, ungepuffert),
- PUTWA/GETWA-Routinen (Random I/O, wortorientiert),
- BUFFERIN/BUFFEROUT in Verbindung mit SETPOS/GETPOS (asynchroner Direct Access I/O).

Für die Beurteilung wurden vor allem folgende Kriterien herangezogen:

- CPU-Zeitverbrauch,
- Anzahl der I/O-Requests pro gelesenem bzw. geschriebenem Record,
- Pufferausnutzung bzw. Verhältnis "angeforderte Daten" zu "tatsächlich übertragene Daten".

Eine Vielzahl von Testbeispielen wurde realisiert, um Leistungsdaten zu ermitteln. Für drei Aufgabenstellungen sollen nachfolgend die Meßergebnisse wiedergegeben werden:

1. Matrixmultiplikation mit zeilenweiser Durchlaufung für Matrizen der Größe 500 × 500. Die bearbeiteten Matrizen wurden hierbei zeilenweise auf den Hintergrundspeicher geschrieben (eine Zeile entspricht einem Record). Ziel dieser Messungen war eine gezielte Beurteilung der Ein-/Ausgabe-Routinen bei rein sequentiellem Schreiben und Lesen. Die Meßergebnisse sind in Tabelle 6 aufgeführt.

2. Zeilenweise Vorwärts-/Rückwärts-Durchlaufung einer 500 × 500-Matrix. Wie bei der Matrixmultiplikation wurde eine Matrixzeile als ein Datenrecord gespeichert. Die Messungen dieses Beispiels sollten insbesondere Aufschluß über die Effizienz der Pufferverwaltung bei nicht-sequentieller Ein-/Ausgabe geben. Die Meßergebnisse finden sich in Tabelle 7.

3. Multiplikation einer 400 × 100-Matrix mit einer 100 × 300 -Matrix mit Hilfe des Software-Paging-Systems. Das Paging-System wurde dabei mit allen zur Verfügung stehenden

Ein-/Ausgabe-Routinen realisiert. Verschiedene Blockzerlegungen wurden getestet. Tabelle 8 gibt die Meßergebnisse für eine Matrixzerlegung in horizontale Blöcke wieder.

I/O-Methode	CPU-Zeit	I/O-Requests	Disk-Sektoren	I/O-Wait	Puffer-größe
Sequentiell	12.73	4195	248828	26.2	16384
Direct Access	16.33	36705	254896	36.3	16384
BUFFER IN/OUT	6.64	3022	248828	13.8	16384
READMS/WRITMS	19.73	19617	247861	33.1	16384
READDR/WRITDR	9.60	253079	253408	1:21.0	0

Tabelle 6: Matrixmultiplikation mit zeilenweiser Durchlaufung

I/O-Methode	CPU-Zeit	I/O-Requests	Disk-Sektoren	I/O-Wait	Puffer-größe
Sequentiell	0.69	5034	43943	7.8	4096
Direct Access	0.63	15503	54547	12.3	4096
BUFFER IN/OUT	0.43	5004	43941	9.4	4096
READMS/WRITMS	0.91	9540	19085	6.5	4096
READDR/WRITDR	0.25	10070	10397	5.1	0

Tabelle 7: Vorwärts-/Rückwärts-Durchlaufung einer Matrix

I/O-Methode	CPU-Zeit	I/O-Requests	Disk-Sektoren	I/O-Wait	Job-größe
Direct Access	1.21	2691	25378	44.6	124416
BUFFER IN/OUT	1.00	2531	24979	22.0	123904
READMS/WRITMS	1.50	2590	25340	55.0	123904
READDR/WRITDR	0.98	2131	25648	57.4	122880
PUTWA/GETWA	1.47	6419	25306	35.2	122880
ohne Paging	0.22	66	407	1.8	275968

Tabelle 8: Matrixmultiplikation mit verschiedenen Varianten des Software-Paging-Systems

Aufgrund der speziellen Problemstellung (Bearbeiten sehr großer Matrizen) wurde davon ausgegangen, daß die zu übertragenden Datenrecords verhältnismäßig groß sein würden (typisch ca. 10000 Worte) und daß der Charakter der Zugriffe eher wahlfrei statt sequentiell sein würde. (Bei einer Blockzerlegung der Matrix wird stets entweder beim Zugriff auf Zeilen oder auf Spalten der Matrix auf nicht-zusammenhängende Datenrecords zugegriffen).

Unter diesen Gesichtspunkten erwiesen sich die Routinen BUFFERIN/BUFFEROUT mit SETPOS/GETPOS einerseits und READDR/WRITDR andererseits als nahezu gleich geeignet.

Alle anderen I/O-Routinen zeigten entweder eine ungünstige Pufferausnutzung beim Datenzugriff oder wiesen sehr hohen CPU-Verbrauch auf.

BUFFERIN/BUFFEROUT zeichnet sich vor allem bei kurzen Datenrecords und bei regelmäßigem (sequentiellem) Datenzugriff durch eine sehr niedrige Anzahl von I/O-Requests und eine gute Pufferausnutzung aus; bei sehr zufälligem Datenzugriff steigt dagegen die Zahl der I/O-Requests an, und je nach Größe der System-I/O-Puffer werden u.U. erhebliche Mengen nicht benötigter Daten transferiert.

READDR/WRITDR realisiert im Gegensatz zu BUFFERIN/BUFFEROUT ungepufferten "direct-to-disk" I/O, d.h. die Daten werden direkt vom externen Speichermedium in die User-Area transferiert. Damit entfällt die Zwischenpufferung im System-I/O-Puffer und die Notwendigkeit der korrekten Wahl der I/O-Puffer-Größe durch den Benutzer. Unabhängig von der Recordlänge wird sowohl bei sequentiellem Datenzugriff als auch bei wahlfreiem Zugriff genau ein I/O-Request pro gelesenem oder geschriebenem Record abgesetzt. Wegen der Direct-to-Disk-Organisation werden die Recordlängen stets auf Vielfache von 512 aufgerundet. Damit eignen sich READDR/WRITDR vor allem für große Recordlängen (> > 512) und wahlfreien Datenzugriff, wie er für die zugrundeliegende Aufgabenstellung typisch ist. Bei der Implementierung des Software-Paging-Systems wurden daher diese I/O-Routinen verwendet.

Ein-/Ausgabe-Routinen im IBM VS-FORTRAN

Im IBM VS FORTRAN stehen im Prinzip zwei Ein-/Ausgabeformen zur Verfügung, die für die Realisierung des Software-Paging genutzt werden können:

- Direct Access I/O,
- Virtual Sequential Access Method (VSAM).

Allerdings ist die Handhabung von VSAM-I/O zumindest im VM/CMS nicht sehr bequem, da spezielle Minidisks zur Abwicklung des I/O verwendet werden müssen. Darüber hinaus haben Vergleichsläufe gezeigt, daß Direct Access I/O eine höhere Effizienz besitzt als VSAM.

Da mit der IBM-Version des Software-Paging außerdem Portabilität angestrebt wird, wurde Direct Access I/O implementiert.

Leistungsvergleich der CRAY- und IBM-Implementierungen mit VDPACK

VDPACK[3] ist ein Unterprogrammsystem, das in ähnlicher Weise wie oben beschrieben eine virtuelle Speicherverwaltung für große Datenbereiche realisiert. Anders als beim oben beschriebenen System unterstützt es nur die Bearbeitung eindimensionaler Datenvektoren. Zwei- oder höherdimensionale Datenstrukturen müssen vom Benutzer durch entsprechende Adreßrechnungen auf eindimensionale Datenbereiche abgebildet werden. Im Gegensatz zum auf der CRAY bzw. IBM implementierten System unterstützt VDPACK eine dynamische Speicherverwaltung, d.h. Datenvektoren können zur Laufzeit in ihrer Größe verändert werden. Bei dem hier vorgestellten Software-Paging-System ist dies nur durch Eröffnen neuer, allerdings beliebig dimensionierbarer Matrizen zur Laufzeit oder durch Wahl einer ausreichend großen Menge an "virtuellem Matrixspeicher" zu Beginn der Rechnung möglich.

---

[3] J. Demel, S. Selberherr: "VDPACK - Ein benutzerorientiertes Unterprogrammpaket zur Realisierung einer dynamischen Speicherverwaltung in FORTRAN"; Angewandte Informatik 6/84, s. 244 ff.; Friedr. Vieweg & Sohn Verlagsgesellschaft mbH.

Auf Einzelheiten der Implementierung von VDPACK soll hier nicht eingegangen werden. Um Anhaltspunkte für einen Leistungsvergleich der beiden Systeme zu geben, sollen hier nur die Leistungsdaten für das Programm SGEFA/SGESL mit Matrizen der Größe $N \times N$ gegenübergestellt werden. Leider standen nur Daten über den Speicherbedarf und den CPU-Zeitverbrauch, nicht jedoch über das I/O-Verhalten von VDPACK zur Verfügung.

Tabelle 9 zeigt die in der zitierten Veröffentlichung angegebenen Daten; sie beziehen sich auf eine Implementierung auf einer CDC CYBER 170/720. Tabelle 10 gibt die entsprechenden Daten für das auf der CRAY X-MP in FORTRAN implementierte System wieder. Tabelle 11 zeigt schließlich die Daten für die auf der IBM 3081 implementierte, portable Version.

FORTRAN-Implementierung von VDPACK					
	ohne VDPACK		FORTRAN VDPACK		Quotient
N	CPU	Speicher	CPU	Speicher	CPU
50	1.0	2600	6.2	8392	6.2
100	3.1	10200	412.2	8592	132.9
150	6.8	22800	2871.9	8792	422.3
200	12.0	40400	7343.0	8992	611.9
250	-	63000	14519.0	9192	-
300	-	90600	26364.4	9392	-
Assembler-Implementierung von VDPACK					
	ohne VDPACK		COMPASS VDPACK		Quotient
N	CPU	Speicher	CPU	Speicher	CPU
50	1.0	2600	3.2	8392	3.2
100	3.1	10200	19.6	8592	6.3
150	6.8	22800	68.0	8792	10.0
200	12.0	40400	147.1	8992	12.3
250	-	63000	278.7	9192	-
300	-	90600	450.9	9392	-

Tabelle 9: Leistungsdaten für VDPACK[3].

Wegen der ungleich größeren Leistungsfähigkeit der CRAY gegenüber der CYBER lassen die absoluten Meßwerte keinen Vergleich der beiden Systeme zu. Allenfalls die durch die Benutzung der Paging-Systeme bedingten relativen Verlangsamungen des Programms können für einen Vergleich herangezogen werden. Dabei muß offen bleiben, welchen Einfluß die relative Leistungfähigkeit der jeweiligen I/O-Systeme der beiden Rechner auf den Leistungsvergleich hat.

Die Blockzerlegung für die CRAY-Implementierung (wie auch für die IBM-Implementierung) wurde so gewählt, daß hauptsächlich der Spaltenzugriff optimiert wird; es wurde allerdings darauf verzichtet, Blöcke mit der Spaltendimension der ganzen Matrix zu verwenden, um den Overhead beim Datentransfer nicht zu groß werden zu lassen.

CRAY FORTRAN-Implementierung						
N	$N_S \times N_Z$	M_K	CPU	Speicher	CPU ohne Paging	Quotient CPU
50	10x10	5	0.1177	613	0.0131	9.0
100	50x25	2	0.2810	2570	0.0498	5.6
150	50x30	3	0.6388	4587	0.1197	5.3
200	50x40	4	1.1413	8100	0.2351	4.9
250	64x32	4	1.8808	8319	0.3903	4.8
300	100x30	3	2.8339	9117	0.6087	4.7

Tabelle 10: Leistungsdaten der CRAY-Implementierung

Um einen dem VDPACK vergleichbaren Speicherplatzbedarf zu erhalten, wurde für $N = 250$ die Matrix durch Hinzufügen von Dummy-Zeilen und -Spalten so erweitert, daß eine Zerlegung in Blöcke der Größe 64×32 möglich ist. Es zeigt sich, daß diese Vergrößerung der Matrix keine merkliche Verschlechterung in der Leistungsfähigkeit des Systems bewirkt.

Ein entscheidender Punkt beim Vergleich von VDPACK mit der CRAY-Implementierung dürfte die Tatsache sein, daß die relative Verlangsamung des Programms bei Verwendung von VDPACK mit steigender Matrixgröße zunimmt, während sie bei der CRAY-Implementierung abnimmt. Dieser Sachverhalt bringt zum Ausdruck, daß das CRAY-Software-Paging-System vor allem für die Bearbeitung sehr großer Matrizen konzipiert ist und bei den Problemgrößen der obigen Beispiele noch unterhalb seiner optimalen Leistungsfähigkeit ist. Matrixgrößen $\geq 1000 \times 1000$ und Blockgrößen ≥ 10000 wurden bei der Implementierung anvisiert.

Portable IBM-FORTRAN-Implementierung						
N	$N_S \times N_Z$	M_K	CPU	Speicher (*4 Byte)	CPU ohne Paging	Quotient CPU
50	10x10	5	1.8034	613	0.1500	12.0
100	50x25	2	9.6933	2570	1.0800	9.0
150	50x30	3	26.9967	4587	3.3567	8.0
200	50x40	4	56.7699	8100	7.7134	7.4
250	64x32	4	104.8333	8316	14.6900	7.1
300	100x30	3	185.5066	9117	25.3033	7.3

Tabelle 11: Leistungsdaten der IBM-Implementierung

Beim Vergleich der IBM-Implementierung mit der CRAY-Implementierung ist zu beachten, daß die Rechengeschwindigkeit der IBM generell deutlich unterhalb der der CRAY liegt. Außerdem wurde die IBM-Software-Paging-Version nicht für die IBM-Anlage optimiert, da für diese Version als wichtigster Gesichtspunkt Portabilität angestrebt wurde. Weiterhin erlaubt die IBM-Version, anders als die CRAY-Variante, das Arbeiten mit unterschiedlichen Datenlängen (4 Byte, 8 Byte und

16 Byte), wodurch die Verarbeitungsgeschwindigkeit etwas vermindert wird. (Wegen der Wortorientierung der CRAY konnten hier alle Fälle einheitlich behandelt werden).

Schlußbemerkungen

Das vorliegende Software-Paging-System wurde speziell für die Benutzung in Verbindung mit Matrix-Algorithmen der Linearen Algebra und verwandter Gebiete auf der CRAY konzipiert. Dabei wurde darauf geachtet, daß die Vektorisierbarkeit solcher Algorithmen nicht zerstört wird. Das Konzept der frei wählbaren Blockzerlegung der zugrundeliegenden Matrix gestattet einen flexiblen Zugriff auf unterschiedlichste Matrixteilstrukturen und gewährleistet hohe Effizienz für sehr unterschiedliche Aufgabenstellungen.

Um Portabilität der Programme zu gewährleisten, wurde außerdem eine weitestgehend maschinenunabhängige, in FORTRAN 77 geschriebene Version zur Verfügung gestellt. Diese Version ist insbesondere auf den IBM-Anlagen des ZAM lauffähig. Anders als die CRAY-Version ist diese Variante nicht maschinenbezogen optimiert. Sie ist daher hinsichtlich der Leistungsdaten mit der CRAY-Version nicht konkurrenzfähig.

Untersuchungen zur Blockzerlegung der Matrizen zeigen, daß eine sorgfältige Wahl von Größe und Gestalt der Blöcke sowie der Größe des Datenpuffers das Paging-Verhalten des Systems sehr nachhaltig beeinflussen können. Testrechnungen mit realen Programmen zeigen, daß bei Zugrundelegen einfacher, idealisierter Zugriffsmuster gefundene theoretische Ergebnisse zur optimalen Blockzerlegung durchaus für die Praxis von Relevanz sein können.

Auch die implementierte Pufferverwaltungsstrategie hat einen gewissen Einfluß auf das I/O-Verhalten. In der vorliegenden Implementierung wurde eine FIFO-Pufferverwaltung gewählt, da sie in allen getesteten Fällen schneller als die vergleichsweise untersuchte LRU-Strategie war und nur unwesentlich mehr I/O verursachte. Es muß zunächst offen bleiben, ob eine speziell für den Datenzugriff bei Matrixalgorithmen konzipierte Pufferverwaltung bessere Resultate erbringen könnte.

Anhang A. Technische Kurzinformation zur Benutzung des Software-Paging-Systems auf der CRAY X-MP

Zweck

Um sehr große Matrizen (ein- oder zweidimensionale Felder) auf der CRAY auch dann bearbeiten zu können, wenn sie nicht vollständig im Hauptspeicher gehalten werden können, wird ein Unterprogrammsystem bereitgestellt, das eine flexible Bearbeitung der Matrizen ohne explizite Benutzung von I/O-Operationen erlaubt.

Das Unterprogrammsystem gestattet Zugriffe auf folgende Matrixteilstrukturen:

- Matrixzeilen (oder Teile davon),
- Matrixspalten (oder Teile davon),
- Diagonalen bzw. Nebendiagonalen (oder Teile davon),
- Matrixelemente,
- Blöcke (rechteckige Teilmatrizen).

Lese- und Schreiboperationen auf der Matrix sind möglich. Bis zu max. 20 Matrizen können gleichzeitig bearbeitet werden (pro Matrix wird eine I/O-Unit benötigt).

Methode

Die zu bearbeitende Matrix wird in Blöcke zerlegt. Die Blöcke sind beliebig rechteckig (z.B. quadratisch oder aus einer oder mehreren Zeilen oder Spalten bestehend). Jeder Block der Matrix wird als ein Record auf dem externen Speicher (Platte oder Buffer Memory) abgelegt. Läßt die vorgegebene Matrix keine geeignete Blockzerlegung zu (z.B. weil die Dimension Primzahl ist), so kann sie durch Zufügen von Dummy-Zeilen und -Spalten so vergrößert werden, daß eine Zerlegung möglich ist.

Alle oben genannten Zugriffsarten (auf Zeilen, Spalten usw.) sind unabhängig von der Blockzerlegung möglich. Allerdings ist die Effizienz der Zugriffe (Anzahl der erforderlichen I/O-Operationen) entscheidend von der gewählten Zerlegung abhängig.

Zur Minimierung der I/O-Operationen wird ein Puffer im Hauptspeicher angelegt, der einen oder mehrere Blöcke aufnehmen kann. Wird beispielsweise auf eine Zeile der Matrix zugegriffen, so werden nach und nach alle Blöcke, die Teile der Zeile enthalten, in den Puffer eingelesen; ist der Puffer groß genug, so verbleiben alle Blöcke im Puffer. Bei nachfolgenden Zugriffen auf benach-

barte Zeilen ist so keine erneute I/O-Operation erforderlich. Werden mehr Blöcke benötigt, als der Puffer aufnehmen kann, so verdrängen neu eingelesene Blöcke die schon am längsten im Puffer befindlichen.

Beispiel: Zerlegung der Matrix A der Dimension N1 x N2 = 300 x 400 in Blöcke der Größe NS x NZ = 100 x 100.

1,1	1,2	1,3	1,4
2,1	2,2	2,3	2,4
3,1	3,2	3,3	3,4

Es ist NBS = N1/NS = 3 die Anzahl der Blöcke pro Spalte und NBZ = N2/NZ = 4 die Anzahl der Blöcke pro Zeile.

Kann der Puffer 6 Blöcke aufnehmen (NBK = 6), so befinden sich nach dem Zugriff auf die 1. Zeile und die 1. Spalte die Blöcke 1,1, 1,2, 1,3, 1,4, 2,1 und 3,1 im Puffer. Die ersten 100 Zeilen und die ersten 100 Spalten können dann ohne weitere I/O-Operationen gelesen und geschrieben werden. Die Verwaltung des Puffers erfolgt automatisch durch das Unterprogrammsystem.

Beschreibung der Unterprogramme

CALL INITBF

(INITialize BuFfer). Initialisiert das Unterprogrammsysytem. INITBF muß einmalig vor dem ersten Aufruf eines der übrigen Unterprogramme gerufen werden.

CALL OPENBF (WKAREA,LEN,UNIT,STATUS,SETVAL,NBK,NS,NZ,NBS,NBZ)

(OPEN BuFfer). Öffnet eine I/O-Unit als Datenträger für eine Matrix. Bis zum Schließen der Unit durch CALL CLOSBF(...) wird die Matrix eindeutig durch diese Unit repräsentiert. Mit dem Aufruf von OPENBF wird die Blockzerlegung der Matrix und ihre Größe festgelegt.

Eingabeparameter:

WKAREA (INTEGER, REAL oder LOGICAL). Eindimensionales Feld der Dimension LEN. WKAREA ist ein Arbeitsbereich, der den Puffer jeder Matrix und einige Indextabellen aufnimmt. Werden mehrere Matrizen gleichzeitig bearbeitet (mehrere Units open), so ist für jede Matrix ein **eigener** Arbeitsbereich bereitzustellen!

Wichtig: Der Speicherplatz für WKAREA muß statisch angelegt sein, d.h. WKAREA sollte in einer SAVE- oder COMMON-Anweisung stehen.

LEN (INTEGER). Dimension des Arbeitsbereichs WKAREA. Es muß gelten

LEN ≥ NBK*NS*NZ + 2*NBS*NBZ + 3*NBK + 48. (s.u.)

UNIT (INTEGER). Nummer der I/O-Unit (0 ≤ UNIT ≤ 99), auf der die Matrix gespeichert wird. Die Unit *nn* ist implizit mit dem temporären Dataset FT*nn* identisch. Der Name des **temporären** Dataset darf nicht geändert werden. Die Matrix kann jedoch für eine spätere Verwendung unter einem beliebigen Namen permanent gespeichert werden.

STATUS (CHARACTER*3). STATUS kann zwei verschiedene Werte annehmen:
'NEW': Die Matrix wird neu angelegt (ein Dataset FT*nn* existiert noch nicht). Beim Anlegen wird die Matrix mit dem Wert 'SETVAL' initialisiert (s.u.).
'OLD': Die Matrix existiert bereits und liegt auf dem Dataset FT*nn*. Es muß die gleiche Blockzerlegung gewählt werden, die beim Anlegen der Matrix spezifiziert wurde (d.h. NS, NZ, NBS und NBZ müssen übereinstimmen; der Wert NBK darf verändert werden). Der Initialisierungswert 'SETVAL' wird ignoriert.

SETVAL (INTEGER, REAL oder LOGICAL). Wert zum Vorbesetzen der Matrix bei STATUS = 'NEW'. Es findet **keine Typkonversion** bei der Initialisierung statt.

NBK (INTEGER). Anzahl der Blöcke im Puffer (vgl. Parameter LEN und obiges Beispiel).

NS (INTEGER). Anzahl der Zeilen pro Matrixblock (1. Index).

NZ (INTEGER). Anzahl der Spalten pro Matrixblock (2. Index).

NBS (INTEGER). Anzahl der Blöcke pro Spalte der Gesamtmatrix.

NBZ (INTEGER). Anzahl der Blöcke pro Zeile der Gesamtmatrix.

Im folgenden bezeichnet der Eingabeparameter UNIT stets die I/O-Unit, auf die sich die Matrixoperation beziehen soll (vgl. OPENBF).

CALL RDROW (UNIT,A,IZ)

(ReaD ROW). Lesen einer vollständigen Zeile der auf der I/O-Unit 'UNIT' gespeicherten Matrix (evtl. einschließlich Dummy-Elemente, die durch Hinzufügen von Dummy-Spalten zur Matrix entstanden sind).

Eingabeparameter:

IZ (INTEGER). Nummer der zu lesenden Zeile (1. Index der Matrix).

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld, das mindestens eine Zeile der Matrix aufnehmen kann (evtl. einschließlich Dummy-Elemente). Enthält nach dem Aufruf die gelesene Matrixzeile. Dimension ≥ NZ * NBZ.

CALL WTROW (UNIT,A,IZ)

(WriTe ROW). Schreiben einer vollständigen Matrixzeile.

Eingabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld, das die zu schreibende Matrixzeile (evtl. einschließlich Dummy-Elemente) enthält. Dimension ≥ NZ * NBZ.

IZ (INTEGER). Nummer der zu schreibenden Zeile (1. Index der Matrix).

CALL RDCOL (UNIT,A,IZ)

(ReaD COLumn). Lesen einer vollständigen Matrixspalte (evtl. einschließlich Dummy-Elemente).

Eingabeparameter:

IZ (INTEGER). Nummer der zu lesenden Spalte (2. Index der Matrix).

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld, das mindestens eine Spalte der Matrix aufnehmen kann (evtl. einschließlich Dummy-Elemente). Enthält nach dem Aufruf die gelesene Matrixspalte. Dimension ≥ NS * NBS.

CALL WTCOL (UNIT,A,IZ)

(WriTe COLumn). Schreiben einer vollständigen Matrixspalte.

Eingabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld, das die zu schreibende Matrixspalte (evtl. einschließlich Dummy-Elemente) enthält. Dimension ≥ NS * NBS.

IZ (INTEGER). Nummer der zu schreibenden Spalte (2. Index der Matrix).

CALL RDPROW (UNIT,A,IZ,J1,J2)

(ReaD Partial ROW). Lesen eines Teils einer Matrixzeile (evtl. einschließlich Dummy-Elemente).

Eingabeparameter:

IZ (INTEGER). Nummer der zu lesenden Zeile (1. Index der Matrix).

J1 (INTEGER). Nummer des ersten zu lesenden Zeilenelements (2. Index der Matrix). Dieses Element wird im ersten Element des Feldes A abgelegt.

J2 (INTEGER). Nummer des letzten zu lesenden Zeilenelements.

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld zum Aufnehmen der gelesenen Teilzeile. Dimension $\geq$ J2-J1 + 1.

CALL WTPROW (UNIT,A,IZ,J1,J2)

(WriTe Partial ROW). Schreiben eines Teils einer Matrixzeile.

Parameter: Analog RDPROW, A ist Eingabeparameter.

CALL RDPCOL (UNIT,A,IZ,J1,J2)

(ReaD Partial COLumn). Lesen eines Teils einer Matrixspalte.

Eingabeparameter:

IZ (INTEGER). Nummer der zu lesenden Spalte (2. Index der Matrix).

J1 (INTEGER). Nummer des ersten zu lesenden Spaltenelements (1. Index der Matrix). Dieses Element wird im ersten Element des Feldes A abgelegt.

J2 (INTEGER). Nummer des letzten zu lesenden Spaltenelements.

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld zum Aufnehmen der gelesenen Teilspalte. Dimension $\geq$ J2-J1 + 1.

CALL WTPCOL (UNIT,A,IZ,J1,J2)

(WriTe Partial COLumn). Schreiben eines Teils einer Matrixspalte.

Parameter: Analog RDPCOL, A ist Eingabeparameter.

CALL RDELEM (UNIT,A,I1,I2)

(ReaD ELEMent). Lesen eines Matrixelements.

Eingabeparameter:

I1	(INTEGER). Erster Index des Matrixelements.
I2	(INTEGER). Zweiter Index des Matrixelements.

Ausgabeparameter:

A	(INTEGER, REAL oder LOGICAL). Einfache Variable, die das gelesene Matrixelement aufnimmt.

CALL WTELEM (UNIT,A,I1,I2)

(WriTe ELEMent). Schreiben eines Matrixelements.

Parameter: Analog RDELEM, A ist Eingabeparameter.

CALL RDDIA (UNIT,A,I1,I2,ND)

(ReaD DIAgonal). Lesen einer Diagonale oder Nebendiagonale oder eines Teils davon.

Eingabeparameter:

I1	(INTEGER). Erster Index des ersten zu lesenden Diagonalelements (= 1 bei Hauptdiagonale).
I2	(INTEGER). Zweiter Index des ersten zu lesenden Diagonalelements (= 1 bei Hauptdiagonale).
ND	(INTEGER). Anzahl der zu lesenden Diagonalelemente. Ist ND größer als die Anzahl der in der Diagonalen enthaltenen Matrixelemente (incl. Dummy-Elemente), so werden nur die in der Diagonalen vorhandenen Elemente gelesen; der Rest des Feldes A bleibt ungeändert.

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Eindimensionales Feld zum Aufnehmen der gelesenen Diagonale. Dimension ≥ ND.

CALL WTDIA (UNIT,A,I1,I2,ND)

(WriTe DIAgonal). Schreiben einer Diagonale oder Nebendiagonale oder eines Teils davon.

Parameter: Analog RDDIA, A ist Eingabeparameter.

CALL RDBLK (UNIT,A,IBS,IBZ)

(ReaD BLocK). Lesen eines Matrixblockes der Größe NS x NZ (wie durch die Blockzerlegung der Matrix definiert).

Eingabeparameter:

IBS (INTEGER). Erster Index des Blocks (vgl. obiges Beispiel).

IBZ (INTEGER). Zweiter Index des Blocks.

Ausgabeparameter:

A (INTEGER, REAL oder LOGICAL). Zweidimensionales Feld der Dimension NS x NZ. Nimmt den gelesenen Matrixblock auf. A ist spaltenweise abgespeichert.

CALL WTBLK (UNIT,A,IBS,IBZ)

(WriTe BLocK). Schreiben eines Matrixblockes der Größe NS x NZ (wie durch die Blockzerlegung der Matrix definiert).

Parameter: Analog RDBLK, A ist Eingabeparameter.

CALL CLOSBF (UNIT,STATUS,IPRT)

(CLOSe BuFfer). Schreiben aller noch im Puffer dieser Unit befindlichen Blöcke auf die I/O-Unit 'UNIT' (sofern die Blöcke verändert wurden) und Schließen der I/O-Unit. Nach Aufruf von

CLOSBF kann der Arbeitsbereich 'WKAREA' (s. OPENBF) für andere Zwecke benutzt werden (z.B. für eine neue Unit).

Eingabeparameter:

STATUS (CHARACTER). STATUS kann zwei Werte annehmen:
'KEEP': Die auf der Unit 'UNIT' (Dataset FT*nn*) gespeicherte Matrix bleibt erhalten. Sie kann permanent gespeichert oder durch erneuten Aufruf von OPENBF mit STATUS = 'OLD' weiter bearbeitet werden.
'DELETE': Der Dataset FT*nn* wird gelöscht. Die Unit 'UNIT' kann nach Aufruf von OPENBF mit STATUS = 'NEW' neu verwendet werden. Evtl. für Unit *nn* bestehende JCL-ASSIGN-Anweisungen werden ebenfalls gelöscht.

IPRT (INTEGER). IPRT kann die Werte 0 oder 1 annehmen:
1: Es wird eine Statistik der insgesamt auf Unit 'UNIT' geschriebenen und gelesenen Records gedruckt.
0: Es wird keine Statistik gedruckt.

Zugriff zum Unterprogrammsystem

Die Unterprogramme stehen in der KFALIB. Sie können bei der Programmausführung durch den LDR-Aufruf

```
LDR,LIB = KFALIB.
```

angezogen werden.

Für Optimierungszwecke steht eine Flowtrace-Version in der KFALIBF zur Verfügung.

Aufruf:

```
LDR,LIB = KFALIBF.
```

Diese Version sollte nur benutzt werden, wenn auch das Hauptprogramm mit ON = F übersetzt wurde.

JCL-Anweisungen zur Dateibehandlung

Falls die zur Speicherung der Matrizen benutzen Dateien temporär sind und keinem speziellen logischen Device zugeordnet werden sollen, werden keine zusätzlichen JCL-Anweisungen zu ihrer Definition benötigt.

Um die I/O-Wait-Zeiten zu verkürzen, ist es jedoch bei Dateien bis zu einer Größe von 3 M-Worten sinnvoll, die Dateien im Buffermemory anzulegen, statt auf Platte. Hierzu ist folgende JCL-Anweisung zu verwenden:

```
ASSIGN,DN = FTnn,DV = BMR.
```

In der Jobkarte muß in diesem Fall durch Angabe von BMR = *blknr* eine ausreichende Menge Speicherplatz im Buffermemory angefordert werden (vgl. "Einführung in die Benutzung der CRAY").

Eine Verwendung des BS-Parameters in der ASSIGN-Anweisung (Festlegen der Größe der I/O-Puffer im Hauptspeicher) ist nicht sinnvoll, da für die Realisierung der I/O-Operationen ungepufferte Ein-/Ausgabe verwendet wird. Eine Verwendung des BS-Parameters bewirkt, daß I/O-Puffer angelegt, aber niemals benutzt werden.

Unter dem lokalen Dateinamen FT*nn* (*nn* = Nummer der I/O-Unit) kann die Datei angesprochen werden, wenn sie permanent gespeichert (SAVE-Anweisung) oder zum Front-End-Rechner geschickt werden soll (DISPOSE-Anweisung). Der lokale Name der Datei darf nicht geändert werden.

Beispiel:

```
DISPOSE,DN=FT55,MF=MC,DF=TR,TEXT='DSN=USRT.ZDV113.MATRIX1,DISP=OLD'.

SAVE,DN=FT55,PDN=MATRIX1,ID=DATEN.
```

Es ist dabei zu beachten, daß die Dateien nur mit dem hier beschriebenen Unterprogrammsystem wieder eingelesen werden können (vgl. OPENBF), da ein spezielles Datenformat verwendet wird. Sollen die Matrizen z.B. im Front-End-Rechner weiter bearbeitet werden, so ist eine vorherige Umspeicherung erforderlich (dies ist z.B. in Verbindung mit den Datenkonvertierungsroutinen möglich).

Empfehlungen zur Blockzerlegung

Allgemein können folgende Empfehlungen zur Blockzerlegung von Matrizen gegeben werden:

- Die Größe des vom Benutzer angelegten Puffers (WKAREA) und die Größe der Matrixblöcke bestimmen wesentlich die Anzahl der erforderlichen I/O-Operationen und ihre Effizienz.

  Der Puffer sollte so groß wie irgend möglich angelegt werden; die Matrixblöcke sollten groß und ihrer Struktur nach dem Zugriffsmuster angepaßt sein (s. Tabelle). Die Größe der Matrixblöcke ist identisch mit der Recordlänge der verwendeten I/O-Operationen.

  Als minimale Blockgröße sollten 10000 Worte angestrebt werden (Vielfache von 512 sind besonders günstig).

- Wird in sehr regelmäßigem Muster zugegriffen (z.B. sequentiell Zeile für Zeile), so sollten die Blöcke groß und nur wenige Blöcke im Puffer sein. Auf diese Weise werden mit jeder I/O-Operation viele benachbarte Zeilen oder Spalten geholt, auf die dann ohne weitere I/O-Operationen zugegriffen werden kann.

- Wird im anderen Extremfall sehr zufällig auf die Matrix zugegriffen, so sollten die Blöcke klein (aber nicht zu klein) und viele Blöcke im Puffer sein. Auf diese Weise werden mit jeder I/O-Operation nur die Daten gelesen bzw. geschrieben, die wirklich benötigt werden. Da viele Blöcke im Puffer sind, ist die Wahrscheinlichkeit, wenigstens Teile von Zeilen, Spalten oder Diagonalen im Puffer zu finden, relativ hoch.

Der Aufwand für I/O-Operationen wird bei sehr zufälligem Zugriff auf die Matrix verhältnismäßig hoch sein.

- Der Wert NBK (Anzahl der Blöcke im Puffer) sollte so gewählt werden, daß mindestens die Blöcke gleichzeitig im Puffer liegen können, die für den Zugriff auf häufig benutzte Matrixteilstrukturen benötigt werden.

  Liegt z.B. eine Zerlegung in 10 mal 10 rechteckige Blöcke vor, so sollten für den Zugriff auf Diagonalen mindestens 19 Blöcke im Puffer Platz finden. Wird bei einer Zerlegung in horizontale Blöcke (1 Block = n Matrixzeilen) alternierend auf die erste und letzte Zeile, dann auf die zweite und vorletzte Zeile usw. zugegriffen, so sollten mindestens 2 Blöcke im Puffer liegen (vgl. Tabelle).

- Eine Optimierung der der Puffer- und Blockgrößen sollte unter Benutzung der Werte "USER I/O-REQUESTS", "DISK SECTORS MOVED" und "I/O-WAIT TIME" aus dem CRAY-Job-Log, sowie der Werte erfolgen, die bei IPRT = 1 (beim CLOSBF-Aufruf) ausgegeben werden.

Die folgende Tabelle gibt Empfehlungen zur Blockzerlegung für einige typische Zugriffsmuster.

Zeilen	Spalten	Diagonalen	Blockzerlegung	NBK
*			Horizontale Blöcke (z.B. eine oder mehrere Zeilen)	≥ 1
	*		Vertikale Blöcke (z.B. eine oder mehrere Spalten)	≥ 1
+	+		Rechteckige Blöcke (möglichst genausoviele Blöcke pro Zeile wie Blöcke pro Spalte)	≥ NBS + NBZ-1
		*	Möglichst quadratische Blöcke	≥ NBS + NBZ-1
+		+	Möglichst quadratische Blöcke	≥ NBS + 2NBZ-2
	+	+	Möglichst quadratische Blöcke	≥ 2NBS + NBZ-2
+	-		Horizontale Blöcke, aber nicht eine ganze Zeile	≥ NBZ
-	+		Vertikale Blöcke, aber nicht eine ganze Spalte	≥ NBS
+	+	+	Möglichst quadratische Blöcke	≥ 2(NBS + NBZ-1)

Dabei bedeutet:

* Ausschließlicher Zugriff
\+ Häufiger Zugriff
\- Sehr seltener Zugriff

Beispiel: Matrixmultiplikation

Gegeben seien drei Matrizen

A: 400x300-Matrix
B: 400x100-Matrix
C: 100x300-Matrix

Zu berechnen sei

A = B * C.

Für die Berechnung auf der CRAY eignet sich das folgende FORTRAN-Programm:

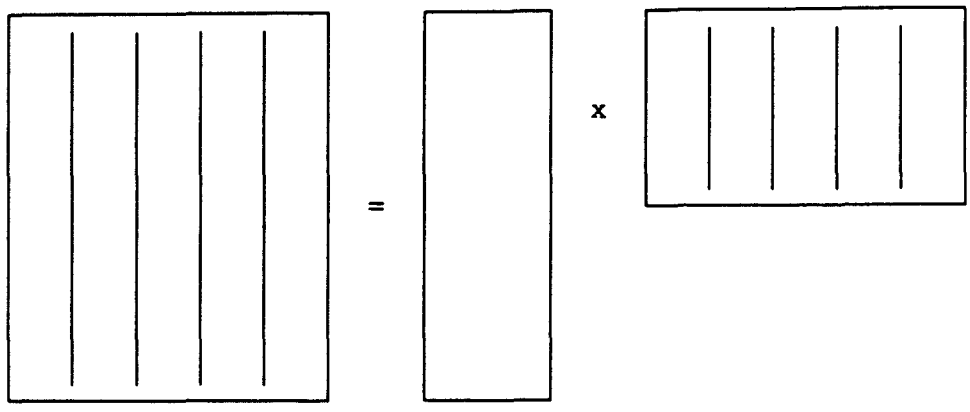
```
      PARAMETER (N1=400,N2=300,N3=100)
C
      DIMENSION A(N1,N2), B(N1,N3), C(N3,N2)
C
C  MATRIX INITIALISIERUNG
C
      DO 1 I = 1,N1
         DO 1 J = 1, N2
              A(I,J) = 0.0
1     CONTINUE
      DO 2 I = 1,N1
         DO 2 J = 1, N3
              B(I,J) = 1.1 + (I/N1) + J/N3
2     CONTINUE
      DO 3 I = 1,N3
         DO 3 J = 1, N2
              C(I,J) = 1.2 + (I+J)/(N2+N3)
3     CONTINUE
C
C  MATRIX MULTIPLIKATION   A = B * C
C
      DO 4 J = 1, N2
         DO 4 K = 1, N3
              DO 4 I = 1, N1
                   A(I,J) = A(I,J) + B(I,K) * C(K,J)
4     CONTINUE
```

Bei Verwendung dieses Programms wird Matrix A einmal spaltenweise, Matrix B N2-mal spaltenweise und Matrix C einmal spaltenweise durchlaufen. (In der innersten Schleife wird die K-te Spalte von Matrix B mit dem Element C(K,J) (K-tes Element der J-ten Spalte von C) multipliziert und elementweise zur J-ten Spalte von Matrix A addiert).

Der Speicherplatzbedarf zum Speichern der Matrizen A, B und C beträgt 190000 Worte, der CPU-Zeitverbrauch (ohne Initialisierung) 0.22 Sekunden.

Um Speicherplatz zu sparen, soll die Matrixmultiplikation mit Hilfe des Unterprogrammsystems realisiert werden. Die Matrizen A und C sollen auf dem externen Speicher gehalten werden. Matrix B bleibt vollständig im Hauptspeicher, da sie sehr häufig vollständig durchlaufen wird und deshalb bei externer Speicherung sehr viel I/O bewirken würde.

Da die Matrizen A und C - beide spaltenweise durchlaufen werden, empfiehlt sich eine Blockzerlegung in Blöcke der Größe NSA x NZA = 400 x 60 für Matrix A bzw. NSC x NZC = 100 x 60 für Matrix C; d.h. jeder Block in Matrix A bzw. Matrix C nimmt 60 Spalten auf. Da beide Matrizen sequentiell Spalte für Spalte durchlaufen werden, brauchen die Puffer jeweils nur einen Block aufzunehmen (NBKA = NBKC = 1).




```
      PARAMETER (N1=400,N2=300,N3=100)
C
C  BLOCKGROESSEN
C
      PARAMETER (NSA=400,NZA=60,   NSC=100,NZC=60)
C
C  BLOCKANZAHL IM BUFFER BZW. PRO ZEILE UND SPALTE
C
      PARAMETER (NBKA=1, NBKC=1)
      PARAMETER (NBSA=1, NBZA=5,    NBSC=1, NBZC=5)
C
C  SPALTENVEKTOREN
C
      DIMENSION COLA(N1), COLC(N3)
      DIMENSION B(N1,N3)
C
C  ARBEITSBEREICHE (PUFFER) FUER MATRIX A UND C
C
      DIMENSION WKAREA (NSA*NZA*NBKA + 2*NBSA*NBZA + 3*NBKA + 48)
      DIMENSION WKAREC (NSC*NZC*NBKC + 2*NBSC*NBZC + 3*NBKC + 48)
C
      INTEGER UNITA, UNITC
C
C  INITIALISIEREN DES UNTERPROGRAMMSYSTEMS
C
      CALL INITBF
C
C  INITIALISIEREN DER PUFFER, I/O-UNITS UND MATRIZEN
C
      UNITA = 11
      UNITC = 13
      SETVAL = 0.0
      LENA = (NSA*NZA*NBKA + 2*NBSA*NBZA + 3*NBKA + 48)
```

```
      LENC = (NSC*NZC*NBKC + 2*NBSC*NBZC + 3*NBKC + 48)
C
      CALL OPENBF(WKAREA,LENA,UNITA,'NEW',SETVAL,NBKA,
     * NSA,NZA,NBSA,NBZA)
      CALL OPENBF(WKAREC,LENC,UNITC,'NEW',SETVAL,NBKC,
     * NSC,NZC,NBSC,NBZC)
C
C  INITIALISIEREN VON MATRIX B UND C
C
      DO  1 J = 1, N3
         DO 1 I = 1,N1
              B(I,J) = 1.1 + (I/N1) + J/N3
1     CONTINUE
      DO 3 J = 1, N2
         DO 2 I = 1,N3
              COLC(I) = 1.2 + (I+J)/(N2+N3)
2        CONTINUE
         CALL WTCOL (UNITC,COLC,J)
3     CONTINUE
C
C  MATRIX MULTIPLIKATION   A = B * C
C
      DO 4 J = 1, N2
         CALL RDCOL (UNITA,COLA,J)
         CALL RDCOL (UNITC,COLC,J)
         DO 5 K = 1, N3
              DO 5 I = 1, N1
                   COLA(I) = COLA(I) + B(I,K) * COLC(K)
5        CONTINUE
         CALL WTCOL (UNITA,COLA,J)
4     CONTINUE
C
C  FREIGEBEN DER PUFFER UND I/O-UNITS
C
      CALL CLOSBF (UNITA,'KEEP',1)
      CALL CLOSBF (UNITC,'DELETE',1)
```

Der Speicherplatzbedarf für die beiden Arbeitsbereiche, die Matrix B und die beiden Spaltenvektoren beträgt bei dieser Realisisierung 70622 Worte; das entpricht einer Reduzierung um den Faktor 2.7. Der CPU-Zeitverbrauch (ohne Initialisierung) beträgt 0.24 Sekunden (Erhöhung um den Faktor 1.1).

Zur Ausführung des Programms ist folgende JCL nötig (die Matrix A wird auf dem Dataset MATRIXA permanent gespeichert):

```
JOB...
ACCOUNT...
CFT.
LDR,LIB=KFALIB.
SAVE,DN=FT11,PDN=MATRIXA,ID=...
EXIT.
/EOF
$INCLUDE PROGRAM FORTRAN A
/EOD
```

Anhang B. Benutzung der portablen Version des Software-Paging-Systems

Grundsätzlich ist die portable Version des Software-Paging-Systems in der gleichen Weise zu benutzen wie die CRAY-Version. Insbesondere gelten alle im Anhang A gemachten Aussagen und Empfehlungen zur Blockzerlegung der bearbeiteten Matrizen. Einige technische Aspekte der Handhabung unterscheiden sich jedoch. Diese sollen im folgenden aufgeführt werden.

Datentypen

IBM-Rechner sind - anders als die CRAY X-MP - Byte-orientierte Maschinen. Daraus resultieren Datentypen mit unterschiedlicher "Wortlänge". Typische Längen sind 4, 8 und 16 Byte. Hieraus entsteht beim Software-Paging die Notwendigkeit, die Wortläge des verwendeten Datentyps als zusätzlichen Parameter beim Öffnen einer "virtuellen" Matrix anzugeben (s.u).

Bei der portablen Version können z. Zt. die Datentypen INTEGER (INTEGER*4), REAL (REAL*4), LOGICAL (LOGICAL*4), DOUBLE PRECISION (REAL*8), COMPLEX (COMPLEX*8), REAL*16 und COMPLEX*16 verwendet werden.

Alle benutzten Variablen und Felder müssen dabei im rufenden Programm diesen Datentyp haben. Dies betrifft insbesondere die Variablen WKAREA und SETVAL der Routine OPENBF sowie alle Ein- und Ausgabefelder (üblicherweise mit A benannt) der übrigen Paging-Routinen (RDROW, WTROW, RDCOL ..., vgl. Anhang A).

Innerhalb einer Matrix sollte der Datentyp fest gewählt sein; werden mehrere Matrizen geöffnet, so dürfen sie unterschiedlichen Typ haben.

Parameterlisten

1. **OPENBF**

   Die Parameterliste lautet

```
OPENBF (WKAREA,LEN,UNIT,STATUS,SETVAL,NBK,NS,NZ,NBS,NBZ,IBYTE).
```

   Die Parameter WKAREA bis NBZ haben die gleiche Bedeutung wie in Anhang A beschrieben. IBYTE gibt die Länge des verwendeten Datentyps in Bytes an. IBYTE = 4, 8 oder 16 sind zulässig.

2. **Übrige Unterprogramme**

Alle übrigen Unterprogramme erhalten einen zusätzlichen Parameter WKAREA. Dieser wird als letzter Parameter der Parameterliste spezifiziert.

WKAREA bezeichnet den Arbeitsbereich, der beim Öffnen der Matrix für das Paging-System zur Verfügung gestellt wurde. Es muß dafür gesorgt werden, daß dieser Bereich in allen Programmteilen, die Unterprogramme des Software-Paging-Systems rufen, mit korrekter Dimensionierung verfügbar ist (z.B. über eine COMMON-Anweisung).

Beispiel:

Die Routine RDROW hat folgende Parameterliste:

```
RDROW (UNIT,A,IZ,WKAREA)
```

Aufruf der portablen Version unter VM/CMS

Die portable Version des Software-Paging-Systems ist z. Zt. nur als Quelltext verfügbar, der bei Bedarf zusammen mit dem Anwenderprogramm übersetzt und geladen werden kann. Um eine einfache Handhabung zu gewährleisten, werden alle erforderlichen CMS FILEDEF-Kommandos zur Laufzeit vom Paging-System beim Öffnen einer Matrix automatisch abgesetzt. (Diese Programmteile sind zu streichen, wenn das System auf einem anderen Rechner benutzt werden soll). Weitere CMS-spezifische Kommandos werden nicht benötigt.

Implementationsabhängige Beschränkungen im VM/CMS

Das Software-Paging-System legt jeden Block der Matrixzerlegung als einen Datenrecord auf dem externen Speichermedium ab. Da bei der portablen Version Direct Access I/O verwendet wird, gelten hiermit für die maximal verwendbare Blockgröße die Limitierungen des VM/CMS bezüglich der maximalen Recordlänge. Die Blockgröße darf deshalb im VM/CMS 32767 Bytes nicht überschreiten.

Anhang C. Beispiele für die Implementierung des Paging-Systems

Initialisieren des Systems (Routine INITBF)

```
C*************************************************************************
C
C   SOFTWARE PAGING FOR CRAY X-MP
C
C   SUBROUTINE PACKAGE FOR HANDLING LARGE 2-DIMENSIONAL MATRICES
C   NOT FITTING INTO MAIN MEMORY BY MEANS OF RANDOM I/O
C   (PACKAGE APPLICABLE ON CRAY ONLY (POINTER STATEMENT AND SPECIAL
C   I/O ROUTINES USED).
C
C                                                VERSION 02 (04/01/85)
C
C                         CHANGES VERSION 00 ==> 01:
C                        - VARIABLE DIMENSION NBK, NS, NZ, NBS, NBZ
C                          STORED TO COMMON ARRAY DIM,
C                        - READDR/WRITDR USED FOR RANDOM I/O
C                        - COLLECTION OF STATISTICS EXTENDED
C
C                         CHANGES VERSION 01 ==> 02:
C                        - DIAGNOSTICS FOR MISSING 'INITBF' IMPROVED
C
C
C             COMPILE WITH OPT=FASTMD:BTREG    !!!
C
C*************************************************************************
      SUBROUTINE INITBF
C*************************************************************************
C
C  INITIALIZE VIRTUAL DATA MEMORY MANAGEMENT FACILITY
C
      IMPLICIT INTEGER (A-Z)
C
      COMMON /HEAP35Q/ KEY,PMS,UNITPT(500),DIM(500)
C
C  INITIALIZE MASTER POINTER FOR HEAP (CONTAINING POINTERS FOR EACH
C     I/O UNIT)
C
      PMS = LOC(UNITPT)
C
      DO 10 I = 1,500
         UNITPT(I) = 0
         DIM(I) = 0
10    CONTINUE
```

```
C
C  SIGNAL 'INITBF HAS BEEN CALLED'
C
      KEY = 123415
      RETURN
      END
```

Initialisieren einer Matrix im virtuellen Speicher (Routine OPENBF)

```
C*********************************************************************
      SUBROUTINE OPENBF (WKAREA,LEN,UNIT,STATUS,SETVAL,NBK,NS,
     *                   NZ,NBS,NBZ)
C*********************************************************************
C
C  OPEN BUFFER FOR VIRTUAL DATA MATRIX HANDLING
C
C  INPUT:
C        WKAREA :  WORK AREA CONTAINING BUFFER STORAGE AND POINTERS
C                  FOR I/O UNIT 'UNIT'
C        LEN    :  LENGTH OF WKAREA
C        UNIT   :  I/O UNIT HOLDING 'VIRTUAL' MEMORY FOR COMPLETE MATRIX
C        STATUS :  'OLD' IF MATRIX ALREADY EXISTS ON DISK, 'NEW' ELSE
C        SETVAL :  VALUE FOR INITIALIZATION OF MATRIX ELEMENTS
C        NBK    :  NUMBER OF BLOCKS CONTAINED IN BUFFER
C                  = DIM (UNIT * 5 + 1)
C        NS     :  FIRST DIMENSION OF MATRIX BLOCK
C                  = DIM (UNIT * 5 + 2)
C        NZ     :  SECOND DIMENSION OF  MATRIX BLOCK
C                  = DIM (UNIT * 5 + 3)
C        NBS    :  NUMBER OF BLOCKS PER COLUMN OF MATRIX
C                  = DIM (UNIT * 5 + 4)
C        NBZ    :  NUMBER OF BLOCKS PER ROW OF MATRIX
C                  = DIM (UNIT * 5 + 5)
C
C
      IMPLICIT INTEGER (A-Z)
C
      CHARACTER*3 STATUS
      CHARACTER*2 CHUN
      CHARACTER*4 FN
      DIMENSION WKAREA(LEN)
C
      COMMON /HEAP35Q/ KEY,PMS,UNITPT(500),DIM(500)
C
      POINTER (PBA,BUFADR(NBS,NBZ))
      POINTER (PXB,BLOCK(NS,NZ))
      POINTER (PBR,BLKRSV(NBK,2))
      POINTER (PUH,PARM(5))
      POINTER (PIN,INDEX(1))
      POINTER (PST,STAT(16,3))
C
C  CHECK FOR CORRECT INITIALIZATION
C
      IF (KEY .NE. 123415) THEN
         WRITE (6,*) 'ERROR IN ROUTINE OPENBF: INITIALIZATION ROUTINE',
     *               ' INITBF HAS NOT BEEN CALLED.'
```

```
         STOP 'PG-ERROR'
      ENDIF
C
      IF ((UNIT .LT. 0) .OR. (UNIT .GT. 99)) THEN
         WRITE (6,*) 'ERROR IN ROUTINE OPENBF: UNIT NUMBER ',UNIT,
     *   ' ILLEGAL'
         STOP 'PG-ERROR'
      ENDIF
      PUH = PMS + UNIT * 5
C
      IF (PARM(1) .NE. 0) THEN
         WRITE (6,*) 'ERROR IN ROUTINE OPENBF: UNIT ',UNIT,
     *   ' ALREADY OPENED.'
         STOP 'PG-ERROR'
      ENDIF
C
      PARM(1) = 1
C
C  SET POINTERS
C
      PBR = LOC(WKAREA)
      PBA = PBR + NBK * 2
      PIN = PBA + NBS * NBZ
      PST = PIN + NBS * NBZ
      PBF = PST + 48
      PXB = PBF
      PARM(2) = PBR
      PARM(3) = PBA
      PARM(4) = PBF
      PARM(5) = PXB
C
C  OPEN UNIT FOR (RANDOM) DIRECT TO DISK I/O
C
      CALL OPENDR (UNIT,INDEX,NBS*NBZ,0)
C
      IF (LEN .LT. NS*NZ*NBK + 2*NBS*NBZ + 3*NBK + 48) THEN
         WRITE (6,*) 'ERROR IN ROUTINE OPENBF: LENGTH OF WORKAREA FOR',
     *   ' UNIT ',UNIT,' TOO SMALL'
         STOP 'PG-ERROR'
      ENDIF
C
C  INITIALIZE BUFADR
C
      DO 10 J = 1, NBZ
      DO 10 I = 1, NBS
         BUFADR(I,J) = 0
10    CONTINUE
C
C  INITIALIZE BLKRSV
C
      DO 20 I = 1, NBK
         BLKRSV(I,1) = 0
         BLKRSV(I,2) = 0
20    CONTINUE
C
C  INITIALIZE STAT (ARRAY FOR COLLECTION OF STATISTICS)
C
      DO 50 I = 1, 16
         STAT(I,1) = 0
```

```
         STAT(I,2) = 0
         STAT(I,3) = 0
50     CONTINUE
C
       STAT (1,1) = STAT(1,1) + 1
C
C
C  STORE ARRAY DIMENSIONS TO COMMON ARRAY 'DIM'
C
       DIM (UNIT*5+1) = NBK
       DIM (UNIT*5+2) = NS
       DIM (UNIT*5+3) = NZ
       DIM (UNIT*5+4) = NBS
       DIM (UNIT*5+5) = NBZ
C
C  INITIALIZE MATRIX ELEMENTS
C
       IF (STATUS .EQ. 'NEW') THEN
          DO 30 I = 1, NZ
          DO 30 J = 1, NS
               BLOCK(J,I) = SETVAL
30        CONTINUE
          DO 40 I = 1, NBS*NBZ
               CALL WRITDR(UNIT,BLOCK,NS*NZ,I,-1,DUMMY)
               STAT(1,3) = STAT(1,3) + 1
40        CONTINUE
       ENDIF
       RETURN
       END
```

Lesen einer Matrixzeile (Routine RDROW)

```
C*******************************************************************************
       SUBROUTINE RDROW (UNIT,A,IZ)
C*******************************************************************************
C
C  READ COMPLETE ROW OF MATRIX STORED ON I/O UNIT 'UNIT'
C
C  INPUT:
C         UNIT    :  I/O UNIT HOLDING 'VIRTUAL' MEMORY FOR COMPLETE MATRIX
C         IZ      :  FIRST INDEX OF MATRIX DETERMINING ROW TO BE READ
C
C  OUTPUT:
C         A       :  ROW OF MATRIX
C
       IMPLICIT INTEGER (A-Z)
       DIMENSION A(1)
C
       COMMON /HEAP35Q/ KEY,PMS,UNITPT(500),DIM(500)
C
C              DIM: BUFADR(NBS,NBZ)
       POINTER (PBA,BUFADR(1))
C              DIM: BLOCK(NS,NZ)
       POINTER (PXB,BLOCK(1))
C              DIM: BLKRSV(NBK,2)
       POINTER (PBR,BLKRSV(1))
```

```
      POINTER (PUH,PARM(5))
C             DIM: WINDOW(NS,NZ)
      POINTER (PWD,WINDOW(1))
      POINTER (PST,STAT(16,3))
C
      PUH = PMS + UNIT * 5
C
      NBK = DIM(UNIT*5+1)
      NS = DIM(UNIT*5+2)
      NZ = DIM(UNIT*5+3)
      NBS = DIM(UNIT*5+4)
      NBZ = DIM(UNIT*5+5)
C
      IF (PARM(1) .NE. 1) THEN
         WRITE (6,*) 'ERROR IN ROUTINE RDROW: UNIT ',UNIT,' NOT OPEN'
         STOP 'PG-ERROR'
      ENDIF
      PBR = PARM(2)
      PBA = PARM(3)
      PBF = PARM(4)
      PST = PBF - 48
      PXB = PARM(5)
C
      STAT(2,1) = STAT(2,1) + 1
C
C  COMPUTE STARTING BLOCK NUMBER
C
      IBS = (IZ-1) / NS + 1
      IRC = (IBS - 1) * NBZ
      IA = 1
      I1 = MOD(IZ-1,NS) + 1
C
      DO 10 IBZ = 1, NBZ
         IF (BUFADR((IBZ-1)*NBS+IBS) .EQ. 0) THEN
              IBK = (PXB-PBF) / (NS*NZ) + 1
              IF (BLKRSV(IBK) .NE. 0) THEN
                   IF (BUFADR((BLKRSV(NBK+IBK)-1)*NBS+BLKRSV(IBK))
     *                   .LT. 0) THEN
C
C                       RESTORE BLOCK TO DISK
C
                        CALL WRITDR(UNIT,BLOCK,NS*NZ,(BLKRSV(IBK)-1)*
     *                  NBZ+BLKRSV(NBK+IBK),-1,DUMMY)
                        STAT(2,3) = STAT(2,3) + 1
                   ENDIF
                   BUFADR((BLKRSV(NBK+IBK)-1)*NBS+BLKRSV(IBK)) = 0
              ENDIF
              CALL READDR (UNIT,BLOCK,NS*NZ,IRC+IBZ)
              STAT(2,2) = STAT(2,2) + 1
C
C             SET BUFADR
C
              BUFADR((IBZ-1)*NBS+IBS) = PXB
              BLKRSV(IBK) = IBS
              BLKRSV(NBK+IBK) = IBZ
C
C             INCREMENT BLOCK POINTER
C
              PXB = PXB + NS * NZ
```

```
            IF (PXB .GE. NS * NZ * NBK + PBF) PXB = PBF
            PARM(5) = PXB
         ENDIF
C
C  READ DATA FROM BUFFER
C
         PWD = ABS(BUFADR((IBZ-1)*NBS+IBS))
         IS = I1
         DO 20 I2 = 1,NZ
            A(IA) = WINDOW(IS)
            IA = IA + 1
            IS = IS + NS
20       CONTINUE
10    CONTINUE
      RETURN
      END
```

Schreiben einer Matrixdiagonale (Routine WTDIA)

```
C*************************************************************************
      SUBROUTINE WTDIA (UNIT,A,I0,J0,ND)
C*************************************************************************
C
C  WRITE (PARTIAL) DIAGONAL OF MATRIX STORED ON I/O UNIT 'UNIT'
C
C  INPUT:
C        UNIT   :  I/O UNIT HOLDING 'VIRTUAL' MEMORY FOR COMPLETE MATRIX
C        A      :  (PARTIAL) DIAGONAL TO BE WRITTEN
C        I0     :  FIRST INDEX OF MATRIX DETERMINING FIRST DIAGONAL
C                  ELEMENT TO BE WRITTEN
C        J0     :  SECOND INDEX OF MATRIX DETERMINING FIRST DIAGONAL
C                  ELEMENT TO BE WRITTEN
C        ND     :  NUMBER OF DIAGONAL ELEMENTS TO BE WRITTEN
C
      IMPLICIT INTEGER (A-Z)
      DIMENSION A(1)
C
      COMMON /HEAP35Q/ KEY,PMS,UNITPT(500),DIM(500)
C
C             DIM: BUFADR(NBS,NBZ)
      POINTER (PBA,BUFADR(1))
C             DIM: BLOCK(NS,NZ)
      POINTER (PXB,BLOCK(1))
C             DIM: BLKRSV(NBK,2)
      POINTER (PBR,BLKRSV(1))
      POINTER (PUH,PARM(5))
C             DIM: WINDOW(NS,NZ)
      POINTER (PWD,WINDOW(1))
      POINTER (PST,STAT(16,3))
C
      PUH = PMS + UNIT * 5
C
      NBK = DIM(UNIT*5+1)
      NS = DIM(UNIT*5+2)
      NZ = DIM(UNIT*5+3)
      NBS = DIM(UNIT*5+4)
```

```
      NBZ = DIM(UNIT*5+5)
C
      IF (PARM(1) .NE. 1) THEN
         WRITE (6,*) 'ERROR IN ROUTINE WTDIA: UNIT ',UNIT,' NOT OPEN'
         STOP 'PG-ERROR'
      ENDIF
C
      PBR = PARM(2)
      PBA = PARM(3)
      PBF = PARM(4)
      PST = PBF - 48
      PXB = PARM(5)
C
      STAT(13,1) = STAT(13,1) + 1
C
      IA = 1
      IX = I0
      JX = J0
C
C  DO FOR ALL POSSIBLE BLOCKS IN DIAGONAL ...
C
      DO 10 IXX = 1, NBS + NBZ
C
C  COMPUTE BLOCK NUMBER AND ELEMENT ADDRESS IN BLOCK
C
         IBS = (IX-1) / NS + 1
         IBZ = (JX-1) / NZ + 1
         IF ((IBS .GT. NBS) .OR. (IBZ .GT. NBZ)) RETURN
         I1 = MOD(IX-1,NS) + 1
         I2 = MOD(JX-1,NZ) + 1
         II = (I2-1)*NS+I1
         IF (BUFADR((IBZ-1)*NBS+IBS) .EQ. 0) THEN
              IBK = (PXB-PBF) / (NS*NZ) + 1
              IF (BLKRSV(IBK) .NE. 0) THEN
                   IF (BUFADR((BLKRSV(NBK+IBK)-1)*NBS+BLKRSV(IBK))
     *                  .LT. 0) THEN
C
C                       RESTORE BLOCK TO DISK
C
                        CALL WRITDR(UNIT,BLOCK,NS*NZ,(BLKRSV(IBK)-1)*
     *                  NBZ+BLKRSV(NBK+IBK),-1,DUMMY)
                        STAT(13,3) = STAT(13,3) + 1
                   ENDIF
                   BUFADR((BLKRSV(NBK+IBK)-1)*NBS+BLKRSV(IBK)) = 0
              ENDIF
              CALL READDR (UNIT,BLOCK,NS*NZ,(IBS-1)*NBZ+IBZ)
              STAT(13,2) = STAT(13,2) + 1
C
C             SET BUFADR
C
              BUFADR((IBZ-1)*NBS+IBS) = PXB
              BLKRSV(IBK) = IBS
              BLKRSV(NBK+IBK) = IBZ
C
C             INCREMENT BLOCK POINTER
C
              PXB = PXB + NS * NZ
              IF (PXB .GE. NS * NZ * NBK + PBF) PXB = PBF
              PARM(5) = PXB
```

```
        ENDIF
C
C  COMPUTE NUMBER OF DIAGONAL ELEMENTS IN BLOCK
C
        NDIA = MIN(NS-I1+1,NZ-I2+1)
        NDX = MIN(NDIA,ND-IA+1)
C
C  WRITE DATA TO BUFFER
C
        PWD = ABS(BUFADR((IBZ-1)*NBS+IBS))
        DO 20 IIX = 1,NDX
             WINDOW(II) = A(IA)
             IA = IA + 1
             II = II + NS + 1
20      CONTINUE
        BUFADR((IBZ-1)*NBS+IBS) = -1 * ABS(BUFADR((IBZ-1)*NBS+IBS))
        IF (IA .GT. ND) RETURN
        IX = IX + NDX
        JX = JX + NDX
10    CONTINUE
      RETURN
      END
```

Schließen einer Matrix (Routine CLOSBF)

```
C*****************************************************************************
      SUBROUTINE CLOSBF (UNIT,STATUS,IPRT)
C*****************************************************************************
C
C  FLUSH BUFFER AND CLOSE I/O UNIT
C
C  INPUT:
C        UNIT   :  I/O UNIT HOLDING 'VIRTUAL' MEMORY FOR COMPLETE MATRIX
C        STATUS :  'KEEP' DATASET FTXX IST KEPT AFTER CLOSE
C                  'DELETE' DATASET FTXX IS DELETED AFTER CLOSE
C        IPRT   :  = 1: STATISTICS ARE TO BE PRINTED, = 0 ELSE
C
      IMPLICIT INTEGER (A-Z)
      CHARACTER*(*) STATUS
      CHARACTER*8 RNAME(16)
      CHARACTER*2 CHUN
      CHARACTER*4 FN
C
      DATA RNAME/'OPENBF  ','RDROW   ','WTROW   ','RDCOL   ',
     *           'WTCOL   ','RDPROW  ','WTPROW  ','RDPCOL  ',
     *           'WTPCOL  ','RDELEM  ','WTELEM  ','RDDIA   ',
     *           'WTDIA   ','RDBLK   ','WTBLK   ','CLOSBF  '/
C
      COMMON /HEAP35Q/ KEY,PMS,UNITPT(500),DIM(500)
C
C             DIM: BUFADR(NBS,NBZ)
      POINTER (PBA,BUFADR(1))
C             DIM: BLOCK(NS,NZ)
      POINTER (PXB,BLOCK(1))
C             DIM: BLKRSV(NBK,2)
      POINTER (PBR,BLKRSV(1))
      POINTER (PUH,PARM(5))
```

```
      POINTER (PST,STAT(16,3))
C
      PUH = PMS + UNIT * 5
C
      NBK = DIM(UNIT*5+1)
      NS = DIM(UNIT*5+2)
      NZ = DIM(UNIT*5+3)
      NBS = DIM(UNIT*5+4)
      NBZ = DIM(UNIT*5+5)
C
      IF (PARM(1) .NE. 1) THEN
         WRITE (6,*) 'ERROR IN ROUTINE CLOSBF: UNIT ',UNIT,' NOT OPEN'
         STOP 'PG-ERROR'
      ENDIF
      PBR = PARM(2)
      PBA = PARM(3)
      PBF = PARM(4)
      PST = PBF - 48
      PXB = PBF
C
      STAT(16,1) = STAT(16,1) + 1
C
C   SAVE REMAINING BLOCKS IF CHANGED
C
      IF (STATUS .NE. 'DELETE') THEN
         DO 10 I = 1, NBK
            IF (BLKRSV(I) .NE. 0) THEN
               IF (BUFADR((BLKRSV(NBK+I)-1)*NBS+BLKRSV(I))
     *             .LT. 0) THEN
                 CALL WRITDR(UNIT,BLOCK,NS*NZ,(BLKRSV(I)-1)*
     *               NBZ+BLKRSV(NBK+I),-1,DUMMY)
                 STAT(16,3) = STAT(16,3) + 1
               ENDIF
            ENDIF
            PXB = PXB + NS*NZ
10       CONTINUE
      ENDIF
C
C  PRINT STATISTICS
C
      IF (IPRT .NE. 0) THEN
         WRITE (6,1004)
1004     FORMAT('1',///,1X,5('*'),' SOFTWARE PAGING ******************',
     *    ' VERSION 02 (04/01/85) ',1('*'),/,' *',T65,'*')
         WRITE (6,1000) UNIT
1000     FORMAT (1X,    '* PAGING STATISTICS:  UNIT ',
     *           I2,T65,'*',/,' *',T65,'*',/,' *',T65,'*')
         WRITE (6,1002)
1002     FORMAT (' *   ROUTINE',3X,' CALLS ',2X,'   READS   ',
     *           '  WRITES     ','  WORDS MOVED',T65,'*',/,
     *           ' *',T65,'*')
         SUMC = 0
         SUMR = 0
         SUMW = 0
         SUMRWW = 0
         DO 30 I = 1, 16
              RWW = (STAT(I,2) + STAT(I,3)) * NS*NZ
              SUMC = SUMC + STAT(I,1)
              SUMR = SUMR + STAT(I,2)
```

```
               SUMW = SUMW + STAT(I,3)
               SUMRWW = SUMRWW + RWW
               WRITE(6,1001) RNAME(I), STAT(I,1),STAT(I,2),STAT(I,3),RWW
1001           FORMAT (' *   ',A8,1X,I7,3X,I8,3X,I8,3X,I15,T65,'*')
30          CONTINUE
C
            WRITE(6,1010) SUMC, SUMR, SUMW, SUMRWW
1010        FORMAT (' *',T65,'*',/
     *             ,' *   TOTAL',4X,I7,3X,I8,3X,I8,3X,I15,T65,'*',/
     *             ,' *',T65,'*')
            WRITE (6,1003) UNIT
1003        FORMAT (' *',T65,'*',/
     *             ,' * UNIT ',I2,' CLOSED',T65,'*',/,' *',T65,'*')
            WRITE (6,1005)
1005        FORMAT(1X,64('*'),///)
         ENDIF
C
C  CLOSE UNIT
C
         IF (STATUS .NE. 'DELETE') THEN
            CALL CLOSDR (UNIT)
         ELSE
            CALL CLOSDR (UNIT)
            WRITE (CHUN,'(I2.2)') UNIT
            FN = 'FT'//CHUN
            CALL RELEASE (DUMMY,'DN',FN)
         ENDIF
         DO 20 I = 1,5
            PARM(I) = 0
20       CONTINUE
         RETURN
         END
```